

Copyright
by
Mengshi Zhang
2019

The Dissertation Committee for Mengshi Zhang
certifies that this is the approved version of the following dissertation:

**An Integrated Approach to Software Testing and
Learning**

Committee:

Sarfraz Khurshid, Supervisor

Christine Julien

Vijay Garg

David Soloveichik

Lingming Zhang

**An Integrated Approach to Software Testing and
Learning**

by

Mengshi Zhang

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2019

Dedicated to my family.

Acknowledgments

I would like to thank all people who helped and supported me for these years.

First of all, I would like to thank my Ph.D. advisor Dr. Sarfraz Khurshid. It is my great fortune to work under his supervision. Sarfraz is a great researcher and advisor, and I learned a lot from his ideas and insights during our discussions; he is a friend, and I appreciate his help. I still remember that he drove me home in his red Dodge at 2 am for catching a paper deadline. Overall, it is quite challenging for me to imagine a better advisor.

I want to thank Dr. Lingming Zhang and Dr. Yuqun Zhang. I enjoyed so much to collaborate with them. I thank them for helping me discuss research ideas and revise papers, and I also appreciate them for their encouragement when I was depressed for some difficulties. I want to thank Dr. Christine Julien, Dr. David Soloveichik and Dr. Vijay Garg, who serve in my qualifying and dissertation committee, for generous help and insightful comments. I want to thank Dr. Milos Gligoric for his helpful comments on my research.

I would like to thank Cong Liu, Corina S. Pasareanu, Jinru Hua, Kaiyuan Wang, Wenxi Wang, Shikhar Singh, Divya Gopinath, Xia Li, Yaoxian Li, Lingchao Chen, Xi Zheng, Dewayne E. Perry. They are my collaborators during my Ph.D.'s study, without their support, the research projects cannot be

successful, and I am looking forward to collaborating them again in the future.

I thank the helpful staff at the Electrical and Computer Engineering Department, especially Melanie Gulick, Melody Singleton, and Cayetana Garcia.

I thank my friends in the Software Engineering track: Chenguang Liu, Changyong Hu, Jinru Hua, Kaiyuan Wang, Wenxi Wang, Shikhar Singh, Nima Dini, Hayes Converse, Xiong Zheng, Chenguang Zhu, Pengyu Nie. I thank my roommate, Jiayuan He, and other friends: Meng Li, Yibo Lin, Xiaodan Xi, Zhuoran Zhao, Shuang Song, Fu Li. I appreciate their support during the past five years.

Finally, I would like to thank my parents for their love, encouragement, and trust, since this work would not have been possible without their support.

This work was funded in part by the National Science Foundation.

An Integrated Approach to Software Testing and Learning

Publication No. _____

Mengshi Zhang, Ph.D.

The University of Texas at Austin, 2019

Supervisor: Sarfraz Khurshid

Software testing – the most commonly used approach for findings bugs – and machine learning – the most popular approach for extracting information from data – are two classic yet disparate approaches that serve largely different purposes and employ fairly unique techniques. Our thesis is that there is a novel bi-directional integration of the approaches that can make each of them more effective. In one direction, we define techniques based on the fundamentals of machine learning to enhance finding and removing bugs in various software systems. In the other direction, we define techniques based on the fundamentals of software testing to enhance the quality of machine learning systems.

Our research consists of three thrusts. One, we define ePRFL, which uses the PageRank algorithm and Method-level Aggregation technique to boost the accuracy of traditional techniques for localizing bugs in code. Two, we define DeepRoad, a GAN-based Metamorphic Testing and Input Validation

framework for DNN-based autonomous driving systems. Three, we define an analysis approach based on symbolic execution to study properties of deep neural networks. We embody the techniques in automated tools and evaluate them using standard benchmarks. The experimental results show the usefulness of our techniques.

Table of Contents

Acknowledgments	v
Abstract	vii
List of Tables	xiii
List of Figures	xv
Chapter 1. Introduction	1
1.1 An Extended Framework for Boosting Spectrum Based Fault Localization via PageRank (TSE'19)	2
1.2 GAN-Based Metamorphic Testing and Input Validation for Autonomous Driving Systems (ASE'18)	4
1.3 Symbolic Execution-Based Attribution and Attack Synthesis for Neural Networks (ISSRE'19, ICSE'19 Poster)	5
Chapter 2. ePRFL	6
2.1 Background	13
2.1.1 Spectrum-Based Fault Localization	13
2.1.2 Link Ranking Algorithms	14
2.1.2.1 PageRank	15
2.1.2.2 HITS and SALSA	18
2.1.3 Method-level Aggregation	20
2.2 Motivating Example	23
2.3 Approach	28
2.3.1 Basic Approach: PRFL	29
2.3.1.1 Preparation Phase: Static and Dynamic Analysis	29
2.3.1.2 Analysis Phase: PageRank Propagation	30
2.3.1.3 Ranking Phase: Weighted Spectrum-Based Fault Localization	34

2.3.2	Extended Approach 1: PRFL+	34
2.3.2.1	PRFL+ Call Graph Refinement	34
2.3.2.2	PRFL+ Passing Test Weight Adjustment	35
2.3.2.3	PRFL+ Test and Method Reduction	36
2.3.3	Extended Approach 2: PRFL _{MA}	38
2.3.4	Computational Complexity	40
2.4	Experimental Setup	42
2.4.1	Subjects	43
2.4.2	Implementation and Supporting Tools/Platform	45
2.4.3	Evaluation Metrics	46
2.5	Result Analysis	48
2.5.1	RQ1: PRFL's Overall Effectiveness and Efficiency	48
2.5.1.1	Effectiveness of PRFL	48
2.5.1.2	Efficiency of PRFL	50
2.5.2	RQ2: Configuration Impacts	53
2.5.3	RQ3: Impact of Fault Number	58
2.5.4	RQ4: Comparison of PRFL and a Recently Proposed SBFL Technique	60
2.5.5	RQ5: Comparison of PRFL, MBFL and the Integrated Approach	62
2.5.6	RQ6: Comparison of the Constrained PageRank and other Link Analysis Algorithms	65
2.5.7	RQ7: Effectiveness of PRFL+	66
2.5.7.1	Impact of Dynamic Call Graph	66
2.5.7.2	Impact of Passing Test Weight Adjustment	67
2.5.7.3	Impact of Test and Method Reduction	69
2.5.8	RQ8: Effectiveness of PRFL _{MA}	75
2.5.9	RQ9: Impact of Fault Type	75
2.5.10	RQ10: Statistical Significance	77
2.6	Threats to Validity	80
2.7	Related Work	82
2.7.1	Spectrum-Based Fault Localization	82
2.7.2	Machine Learning-Based Fault Localization	83

2.7.3	Mutation-Based Fault Localization	84
2.7.4	Slicing-Based Fault Localization	84
2.7.5	Other Fault Localization Techniques	85
Chapter 3.	DeepRoad	87
3.1	Background	94
3.1.1	DNN Architectures	94
3.1.1.1	Convolutional Neural Networks	95
3.1.1.2	Recurrent Neural Networks	95
3.2	Approach	98
3.2.1	Metamorphic Testing for DNN-Based Autonomous Driving Systems	98
3.2.1.1	Metamorphic DNN Testing	98
3.2.1.2	DNN-Based Road Scene Transformation	99
3.2.1.3	Framework of DeepRoad _{MT}	102
3.2.2	Input Validation for DNN-Based Autonomous Driving Systems	103
3.2.2.1	Input Validation of DNNs	104
3.2.2.2	Framework of DeepRoad _{IV}	106
3.3	Experiments	110
3.3.1	Data	110
3.3.2	Models	111
3.3.3	Metric	112
3.3.4	Results	114
3.3.4.1	Results of DeepRoad _{MT}	114
3.3.4.2	Results of DeepRoad _{IV}	118
3.4	Threats to Validity	122
3.5	Related work	125
3.5.1	Metamorphic Testing	125
3.5.2	Input Validation	125
3.5.3	Testing and Verification of DNN-Based Autonomous Driving Systems	126
3.5.4	GAN-Based Image Translation	126

Chapter 4. DeepCheck	128
4.1 Background: Neural Networks	134
4.2 Overview	135
4.2.1 Identifying Important Pixels	135
4.2.2 Identifying Attack Pixels	137
4.3 Technique	140
4.3.1 Symbolic Execution Framework	140
4.3.2 Identifying Important Pixels	142
4.3.3 Synthesizing Attacks	148
4.4 Evaluation	150
4.4.1 Subject Neural Network and Images	150
4.4.2 Important Pixel Identification	150
4.4.3 t -pixel Attack	151
4.4.4 Important Pixels and Attackable Pixels	156
4.5 Related Work	159
4.5.1 Techniques for Attribution	160
4.5.2 Techniques for Adversarial Attack Generation	161
Chapter 5. Conclusion	163
Bibliography	164

List of Tables

2.1	Spectrum-based Fault Localization Techniques and Definitions	14
2.2	Original and Weighted Spectra of Code . (T denotes Tarantula score)	27
2.3	The Process of $PRFL_{MA}$	39
2.4	Subject Statistics	44
2.5	Results of SBFL and PRFL on All Defects4J Faults	49
2.6	Fault Localization Overheads	50
2.7	Results of SBFL and PRFL on All Bugs.jar Faults	51
2.8	Average Memory Consumption of PRFL	52
2.9	Overall fault Localization Results on Single-fault and Multi-location-faults of Defects4J	59
2.10	Effectiveness of Ochiai, Multric and PRFL	61
2.11	Overall Results of the Comparison of PRFL, MBFL and Their Combination	64
2.12	Overall Results of the Comparison of Link Analysis Algorithms	65
2.13	Overall Results of Dynamic Call Graph	67
2.14	Results of Passing Test Weight Adjustment	68
2.15	Result Snippet of Math-1	69
2.16	Results of Test-based Reduction	71
2.17	Results of Method-based Reduction	72
2.18	Average Test (Left) and Method (Right) Number after Reduction	73
2.19	Computation Time of PRFL, Test-based Reduction and Method-based Reduction	74
2.20	Result of SBFL,PRFL,SBFL _{MA} and PRFL _{MA} on all Defects4J bugs	78
2.21	Fault localization Results on Mutation Faults	79
2.22	Paired <i>t</i> -test Results of Different Techniques	79
3.1	Details of image sets	110

3.2	Number of inconsistency behavior of three models under different weather conditions	118
4.1	Top-5% and Top-10% of Important Pixels (Green) Identified by DeepCheck ^{Imp} for <i>abs</i> , <i>co</i> , and <i>coi</i>	152
4.2	Top-30% (i.e., 235) of Important Pixels (Green) Identified by DeepCheck ^{Imp} for <i>abs</i> , <i>co</i> , and <i>coi</i>	153
4.3	Number of 1-pixel Attackable Pixels (#ap) Identified by DeepCheck ^{tPA} for Each Image (digit) among All Pixels. “alabel” shows the set of labels that the network incorrectly produces after the 1-pixel attacks. “1stap” shows the smallest position (pixel ID) of any attackable pixel in the image. For digit 6, the pixel at position 486 can be attacked to become 5 as well as 8 by appropriately setting the pixel’s value.	154
4.4	Number of 2-pixel Attacks (#a2p) Identified by DeepCheck ^{tPA} for Each Image (digit), and of These Attacks the Number that do not Use Any Pixel that is Attackable for a 1-pixel Attack (#a2p-new). “alabel” shows the set of labels that each digit can be attacked to incorrectly produce using the neural net. The search for 2-pixel attacks is focused on top-5% important pixels identified by the <i>coi</i> metric of DeepCheck ^{Imp}	154
4.5	Important Pixels and Attackable Pixels	155
4.6	Smallest Number of Important Pixels to Explore for the First 1-pixel Attack for Each Image (Digit) and Each Metric	157
4.7	Smallest Number of Important Pixels to Explore for the First 2-pixel Attack for Each Image for the <i>coi</i> Metric	157

List of Figures

1.1	Overview of my Ph.D. work	2
2.1	A Small Network and its Transition Matrix	16
2.2	Example Code Snippet for Method-level Aggregation	21
2.3	Example Code and Corresponding Test Suite	24
2.4	Test Coverage Graph of Code and the Transition Matrix of Failing Tests. (Red means the tests that failed.)	25
2.5	Framework of the Proposed Approach	28
2.6	Test-based Reduction (a) and Method-based Reduction (b)	37
2.7	Impact of Damping Factor	54
2.8	Impact of Call Graph Weight	55
2.9	Impact of Return Weight	56
3.1	Driving Scenes Synthesized by DeepXplore (a)(b) and DeepTest (c)(d)(e)	89
3.2	Snowy and Rainy Scenes Synthesized by DeepRoad	89
3.3	Autonomous Driving Systems Built on CNN and RNN	96
3.4	Structure of UNIT	101
3.5	Framework of DeepRoad _{MT}	102
3.6	Framework of DeepRoad _{IV}	108
3.7	Images Collected from YouTube	111
3.8	Real and GAN-Generated Images	111
3.9	Results of Testing Mode Collapse of UNIT Models	115
3.10	Inconsistency of Steering Angle Prediction on Original and Synthetic Images	116
3.11	Results of DeepRoad _{IV} : Image Embeddings and Distance Distributions	119
3.12	Results of DeepRoad _{IV} (without non-linear Transformation): Image Embeddings and Distance Distributions	120

3.13	Results of Differential Testing	122
4.1	(a) Example Image with Predicted Label 3. (b) Top-5% and (c) Top-10% Important Pixels (Green) Identified by DeepCheck ^{Imp} ; (d) 1-pixel Attack (Red) Identified by DeepCheck ^{tPA} , Chang- ing the Red-pixel to black Changes the Predicted Label to 8; (e) 2-pixel Attack (Red) that Does Include an Attackable Pixel for 1-pixel Attack.	135
4.2	Branching Transformation for Neuron i	141
4.3	Updating the <i>coef</i> Arrays. ($\cdot$ is the dot-product of vectors). . .	145
4.4	Ten Images from the MNIST Training Dataset [104]	151
4.5	Attackable Pixels for 1-pixel Attack Highlighted in Red	151
4.6	Attackable Pixels for 2-pixel Attack Highlighted in Red. For each digit, the union of all pixels that are part of any 2-pixel attack is shown.	155

Chapter 1

Introduction

The basis of this work is two common and well-studied approaches, namely software testing [22] and machine learning [29], which have co-existed for several decades but evolved largely independently. Software testing is the most commonly used approach for finding bugs, in particular, and validating quality of software, in general. Machine learning is the most popular approach for extracting information from data, and in recent years has seen an unprecedented increase in its use across a variety of domains. Traditionally, software testing and machine learning are two disparate approaches that serve largely different purposes and employ fairly unique techniques.

Our thesis is that there is a novel bi-directional integration of the approaches that can make each of them more effective. In one direction, we define techniques based on the fundamentals of machine learning to enhance finding and removing bugs in various software systems. In the other direction, we define techniques based on the fundamentals of software testing to enhance the quality of machine learning systems.

Our research consists of three thrusts. One, we define ePRFL, which uses the PageRank algorithm to boost the accuracy of traditional techniques

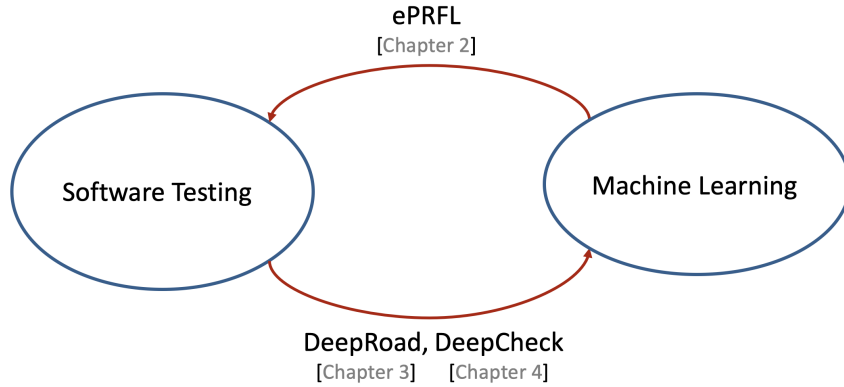


Figure 1.1: Overview of my Ph.D. work

for localizing bugs in code [169,170] (Section 1.1). Two, we define DeepRoad, a GAN-Based Metamorphic Testing and Input Validation framework for DNN-based autonomous driving systems [172] (Section 1.2). Three, we define an analysis approach based on symbolic execution to study the properties of deep neural networks [62,63] (Section 1.3). We embody the techniques in automated tools and evaluate them using standard benchmarks. The experimental results show the usefulness of our techniques.

1.1 An Extended Framework for Boosting Spectrum Based Fault Localization via PageRank (TSE’19)

Localizing bugs and removing them – two key parts of *debugging* code – are notoriously tedious and time-consuming. Therefore, various automated techniques have been proposed to help with manual debugging. Among the existing fault localization techniques, *spectrum-based fault localization* (SBFL) is one of the most widely studied techniques due to being lightweight [72].

A focus of the existing SBFL techniques is to consider how to differentiate program entities (i.e., one dimension in program spectra); indeed, this focus is aligned with the ultimate goal of finding the faulty lines of code. Our key insight is to enhance the existing SBFL techniques by additionally considering how to differentiate tests (i.e., the other dimension in program spectra) [169, 170].

In Chapter 2, we introduce ePRFL, an enhanced version of PRFL [169, 170] which is a lightweight technique that boosts SBFL by differentiating tests using PageRank algorithm. Given the original program spectrum information, PRFL uses PageRank to *recompute* the spectrum by considering the contributions of different tests. Then, traditional SBFL techniques can be applied on the recomputed spectrum to achieve more effective fault localization. ePRFL enhances PRFL by altering/optimizing its components and combining it with the Method-level Aggregation technique as a new PRFL variant. Though being simple and lightweight, PRFL has been demonstrated to outperform state-of-the-art SBFL techniques significantly (e.g., ranking 39.2% / 82.3% more real/artificial faults within Top-1 compared with the most effective traditional SBFL technique) with low overhead (e.g., around 6 minutes average extra overhead) on 395 real faults from 6 Defects4J projects and 96925 artificial (i.e., mutation) faults from 240 GitHub projects, demonstrating a promising future for considering the contributions of different tests during fault localization. Additionally, ePRFL can locate 137 real faults (73.4% / 24.5% more compared with the most effective SBFL/PRFL technique) and 35058 artificial

faults (159.6% / 28.1% more than SBFL/PRFL technique) at Top-1.

1.2 GAN-Based Metamorphic Testing and Input Validation for Autonomous Driving Systems (ASE’18)

While Deep Neural Networks (DNNs) have established the fundamentals of image-based autonomous driving systems, they may exhibit erroneous behaviors and cause fatal accidents. To address the safety issues in autonomous driving systems, a recent set of testing techniques have been designed to automatically generate artificial driving scenes to enrich test suite, e.g., generating new input images transformed from the original ones. However, these techniques are insufficient due to two limitations: first, many such synthetic images often lack the diversity of driving scenes, and hence compromise the resulting efficacy and reliability. Second, for machine-learning-based systems, a mismatch between training and application domain can dramatically degrade system accuracy, such that it is necessary to validate inputs for improving system robustness.

In Chapter 3, we introduce DeepRoad, an unsupervised DNN-based framework for automatically testing the consistency of DNN-based autonomous driving systems and validating input images. First, DeepRoad automatically synthesizes large amounts of diverse driving scenes without using image transformation rules (e.g. scale, shear, and rotation). In particular, DeepRoad can produce driving scenes with various weather conditions (including those with rather extreme conditions) by applying Generative Adversarial Networks (GANs)

along with the corresponding real-world weather scenes. Second, DeepRoad utilizes metamorphic testing techniques to check the consistency of such systems using synthetic images. Third, DeepRoad validates input images for DNN-based systems by measuring the distance of the input and training images using their VGGNet features. We implement DeepRoad to test three well-recognized DNN-based autonomous driving systems in Udacity self-driving car challenge. The experimental results demonstrate that DeepRoad can detect thousands of inconsistent behaviors for these systems, and effectively validate input images to potentially enhance the system robustness as well.

1.3 Symbolic Execution-Based Attribution and Attack Synthesis for Neural Networks (ISSRE’19, ICSE’19 Poster)

In Chapter 4, we introduce DeepCheck for validating DNNs based on core ideas from program analysis, specifically from symbolic execution. The idea is to translate a DNN into an imperative program, thereby enabling program analysis to assist with DNN validation. A basic , however, creates programs that are very complex to analyze. DeepCheck introduces novel techniques for lightweight symbolic analysis of DNNs and applies them in the context of image classification to address two challenging problems in DNN analysis: 1) the identification of important pixels (for attribution and adversarial generation), and 2) the creation of 1-pixel and 2-pixel attacks.

Chapter 2

ePRFL: An Extended Framework for Boosting Spectrum Based Fault Localization via PageRank¹

Software debugging is an expensive and painful process that costs developers a lot of time and effort. For example, it has been reported that debugging can take up to 80% of the total software cost [148, 154]. Thus, there is a pressing need for automated debugging techniques. In the last two decades, various fault localization approaches have been proposed to help developers locate the root causes of failures, e.g., spectrum-based [18, 41, 72, 109, 157, 164], slicing-based [100, 160], machine-learning-based [51, 159], and mutation-based [94, 105, 114, 168] techniques. The recent survey by Wong et al. [154] shows more details about various fault localization approaches.

Among the existing fault localization approaches, spectrum-based fault localization (SBFL), is one of the most widely studied fault localization techniques in the literature [98, 109, 157, 164, 167]. Despite that SBFL is a particu-

¹This chapter is based on *An Empirical Study of Boosting Spectrum-based Fault Localization via PageRank* [171]. This work is collaborated with Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang and Sarfraz Khurshid. Mengshi Zhang is the major contributor of this work, who purposed ideas, designed experiments and wrote the main sections of the paper.

larly lightweight approach, it has been shown to be competitive compared to other approaches [120]. SBFL techniques take as input a set of passing and failing tests, and analyze program execution traces (spectra) of successful and failed executions. The execution traces record the program entities (such as statements, basic blocks, and methods) executed by each test. Intuitively, a program entity covered by more failing tests but less passing tests is more likely to be faulty. Hence, SBFL applies a ranking formula to compute suspiciousness scores for each entity based on the program spectra. Suspiciousness scores reflect how likely it is for each program entity to be faulty, and can be used to sort program entities. Then, developers can follow the suspiciousness rank list (from the beginning to the end of the list) to manually inspect source code to diagnose the actual root causes of failures. Recently, SBFL techniques have also been utilized by various automated program repair techniques to localize potential patch positions [46, 54, 87, 97, 101, 128, 158].

The advantage of SBFL is quite obvious – it is an extremely lightweight approach that is scalable and applicable for large-scale programs. An ideal fault localization technique would always rank the faulty program entities at the top. However, in practice, although various SBFL techniques have been proposed (such as Jaccard/Ochiai [18], Op2 [109], and Tarantula [72]), no technique can always perform the best – the developers usually have to check various false-positive faults before finding the real one(s). We believe the current form of spectrum analysis is a key reason that limits the effectiveness of all existing SBFL techniques. Although different existing SBFL techniques use different

formulas for suspiciousness computation, they all only consider how to differentiate *program entities* (i.e., *one dimension in program spectra*). Our key insight is that a richer form of spectrum analysis that additionally considers how to differentiate *tests* (i.e., *the other dimension in program spectra*), which, to the best of our knowledge, has not been studied in previous work, can provide more effective fault localization. For instance, consider two tests **t1** and **t2** that are both failing tests, where **t1** covers 100 program entities and **t2** only covers one. By our intuition, **t2** is much more helpful than **t1** in fault localization since **t2** has a much smaller search space to localize the fault(s). However, the traditional SBFL techniques ignore this useful information and consider **t1** and **t2** as making the same contribution on SBFL, e.g., a program entity executed by **t1** or **t2** once will be treated the same regardless of the number of entities covered by the tests.

To overcome the limitations of the existing SBFL techniques, we utilize the existing program spectra more effectively by explicitly considering the contributions of different tests. Based on our insight, we present PRFL [170], a lightweight PageRank-based technique that boosts SBFL by considering the additional test information via PageRank algorithm [112]. PRFL captures the connections between tests and program entities (e.g., the traditional spectrum) as well as the connections among program entities (e.g., the static call graphs) via bytecode instrumentation and analysis. Then, PageRank is used to recompute the program spectra: (1) program entities connected with more important failing tests (which cover a smaller number of program entities) may be more

suspicious, and (2) program entities connected with more suspicious program entities may also be more suspicious since they may have propagated the error states to the connected entities. Finally, PRFL employs existing SBFL ranking formulas to compute the final suspiciousness score for each program entity. We have used our PRFL prototype to localize the faulty methods for 395 real faults in Defects4J [73] benchmark. Since mutation faults have also been shown to be suitable for software testing experimentation [24, 74], to further validate the effectiveness of the proposed approach, we apply it to localize 96925 artificial mutation faults generated from 240 GitHub Java projects. The experimental results demonstrate that our technique can outperform state-of-the-art SBFL techniques significantly (e.g., ranking 39%/103% more real/artificial faults as Top-1) with negligible overhead (e.g., around 6 minutes average overhead on real faults).

To enhance our prior work [170], we extend the evaluation in three aspects to further validate the effectiveness of our approach. First, we compare PRFL with a recently proposed learning-based SBFL technique (i.e., Multic [159]). Second, we compare PRFL with two recently proposed mutation-based fault localization (MBFL) techniques (i.e., Metallaxis [116], and MBFL-hybrid-avg [120]). Moreover, we study a simple integration of PRFL and MBFL, and compare its performance with PRFL and MBFL. Third, we study the performance of PRFL using three different link analysis techniques (i.e., standard PageRank [112], HITS [81] and SALSA [90]). The experimental results demonstrates that PRFL still outperforms the compared and integrated

approaches.

On the other hand, to improve the efficacy of PRFL, we further extend the study and propose PRFL+. To be more specific, first, in PRFL, the static call graph analysis is applied to reflect the connections among program entities for simplicity. However, it might not be accurate to reflect the actual run-time connections. Hence, we study the effectiveness of PRFL by using dynamic call graphs to collect the runtime information. Second, PRFL originally assigns the same weight to all passing tests though their test scopes (i.e., numbers of covered program entities) are different. We extend the work by modifying the weights of passing tests to investigate their influence. Third, it is observed that some tests and methods make a negligible contribution for fault localization in PRFL. To reduce the computational complexity of PRFL, we add a coverage-refinement module to remove the irrelevant methods and tests prior to PageRank analysis.

Furthermore, we propose an aggregation and PageRank-based fault localization technique, PRFL_{MA}, which applies Method-level Aggregation [142] to PRFL. The Method-level Aggregation first derives the program spectra of all statements within a method and applies SBFL formula to calculate their suspiciousness scores. Next, for each method, it chooses the most suspicious statement and uses its coverage to substitute the coverage of the given method. Finally, PRFL_{MA} executes PRFL on the updated coverage to rank all the methods. We also evaluate PRFL_{MA} on the Defects4J benchmark. The experimental results show that PRFL_{MA} successfully locates 73.4%/24.5% more real

faults than state-of-the-art SBFL/PRFL techniques and 159.6%/28.1% more artificial faults than the state-of-the-art SBFL/PRFL techniques. Moreover, to study the generalizability of PRFL, we evaluate our approach on Bugs.jar benchmark [131], and the result shows PRFL can rank 30% more faults as Top 1.

Overall, we make the following contributions [170]:

- **Novel Idea.** We propose a novel idea that considers the different contributions of different tests to further boost SBFL.
- **Lightweight Technique.** We implement the proposed idea as a lightweight fault localization technique, PRFL, that uses PageRank to consider the weights of different tests to enhance SBFL.
- **Extensions.** We study our PRFL technique by replacing static call graphs with dynamic call graphs, adjusting the modeled weights of passing tests, and reducing the size of tests and methods. Moreover, we propose PRFL_{MA} which utilizes statement-level information to improve the fault localization accuracy of PRFL.
- **Practical Tool.** We implement all the proposed techniques (PRFL and PRFL_{MA}) as a practical Maven Plugin, *intelliFL*, which is publicly available on Maven Central Repository, and supports Java programs with JUnit tests developed under a variety of JDK and JUnit versions.

- **Evaluation.** We evaluate our PRFL and its extensions on both real and artificial faults. First, we evaluate our approach on 395 real faults from all 6 projects in the Defects4J benchmark. To reduce the threats to external validity, we further evaluate PRFL and PRFL_{MA} on 96925 mutation faults of 240 GitHub projects. The experimental results demonstrate that PRFL and PRFL_{MA} can outperform the state-of-the-art SBFL techniques significantly (e.g., ranking 39% and 73% more faults as Top-1 on Defects4J benchmark, respectively). Moreover, we further compare PRFL to the recently proposed fault localization techniques, study the integration of PRFL and MBFL, and investigate the impact of different link analysis approaches on PRFL. The experimental results show that PRFL still outperforms the compared and integrated approaches. At last, we study the generalizability of PRFL on another benchmark Bugs.jar [131], and the result shows PRFL can help locate around 30% more faults at Top 1.

2.1 Background

2.1.1 Spectrum-Based Fault Localization

Spectrum-based fault localization techniques (SBFL) [18, 72, 109, 153] help developers identify the locations of faulty program entities (such as statements, basic blocks, and methods) based on observations of failing and passing test executions. An SBFL technique sorts all program entities by their suspiciousness scores and returns a rank list for manual checking. If a program entity is more likely to be faulty, it will be assigned a higher priority in the suspiciousness list. Therefore, an ideal SBFL technique should always rank the faulty entity with high suspiciousness score, which can significantly speed up the debugging process for finding the root causes of test failures. To compute suspiciousness scores of program entities, an SBFL technique firstly runs tests on the target program and records the program spectrum of each failing or passing test, i.e., the run-time profiles about which program entities are executed by each test. Then, based on the program spectra and test outcomes, various statistics can be extracted for suspiciousness computation, e.g., tuple (e_f, e_p, n_f, n_p) , where e_f and e_p are the numbers of failing and passing tests executing the program entity e , while n_f and n_p are the numbers of failing and passing tests that do not execute e . Based on such tuples, various SBFL formulas have been proposed. The common intuition of these formulas is that a program entity executed by more failing tests and less passing tests is more likely to be faulty. This work considers 8 well-studied SBFL techniques – Tarantula, Statistical Bug Isolation (SBI), Ochiai, Jaccard, Ochiai2,

Tech	Defn	Tech	Defn
Tarantula	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f} + \frac{e_p}{e_p+n_p}}$	SBI	$1 - \frac{e_p}{e_p+e_f}$
Ochiai	$\frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}}$	Jaccard	$\frac{e_f}{e_f+e_p+n_f}$
Ochiai2	$\frac{e_f n_p}{\sqrt{(e_f+e_p)(n_f+n_p)(e_f+n_p)(n_f+e_p)}}$	Kulczynski	$\frac{e_f}{n_f+e_p}$
Op2	$e_f - \frac{e_p}{e_p+n_p+1}$	Dstar2	$\frac{e_f^2}{e_p+n_f}$

Table 2.1: Spectrum-based Fault Localization Techniques and Definitions

Kulczynski, Op2, and Dstar2 [18, 72, 95, 109, 153]. Tarantula, SBI, Ochiai and Jaccard [159, 160] are the most widely-used techniques for the evaluation of fault localization. Ochiai2 [109] is an extended version of Ochiai, which considers the impact of non-executed or passing test cases. Op2 [109] is the optimal SBFL technique for single-fault program, whereas Kulczynski and Dstar2 belong to the formula family Dstar [153], which is shown to be more effective than 38 other SBFL techniques. All their formulas are listed in Table 2.1.

2.1.2 Link Ranking Algorithms

Ranking plays an important role in information retrieval systems. In particular, due to the rapid growth of the World Wide Web, an efficient search engine is expected to rank and recommend web pages corresponding to users' preference to significantly save users' time and effort to find the web pages which they are interested in. A straightforward example is that we can utilize word frequency to rank web pages according to their relevance to user's query. However, this approach may rank more web pages with low authority to the top

since they have stronger text relevance. One solution to address this issue is to use the information of the Web structure instead of text similarity only. In the recent two decades, several link ranking algorithms were invented to efficiently extract structural information [31, 49]. Generally, the link ranking algorithms can be classified into query independent algorithms (e.g., PageRank [112]) and query dependent algorithms (e.g., HITS [81] and SALSA [90]) that are specifically introduced as follows.

2.1.2.1 PageRank

PageRank [112] is proposed by Larry Page and Sergey Brin for improving search quality and speed. PageRank views the World Wide Web as a set of linked nodes and ranks them based on their importance. The intuition behind PageRank is, for each node, if it is linked by important nodes, it should be more important than the ones linked by uninfluential nodes. Figure 2.1 presents a simple directed graph to describe a small network with four web pages, denoted by node A , B , C and D . The edges between two nodes denote that the starting node contains a hyperlink pointing to the ending node.

By our observation, the number of edges pointing to D is larger than others, so it should be more important than others. On the other hand, A is pointed by D and thus is also an important node according to the assumption of PageRank. Then, B is in turn pointed by A , and also should be assigned a high score to show the importance. Formally, the websites are described by a directed graph $G = \langle V, E \rangle$ with n nodes and m edges. Let

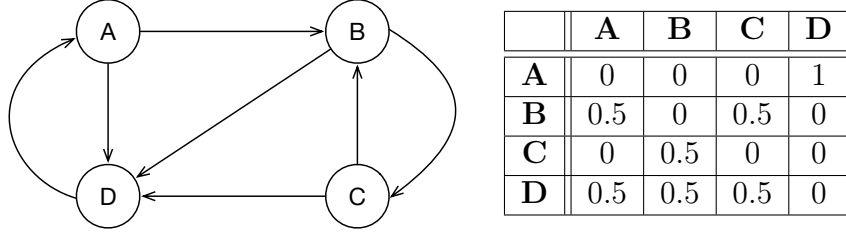


Figure 2.1: A Small Network and its Transition Matrix

$\mathbf{P}$ be the transition matrix of n by n elements. Then, each matrix element, P_{ij} , denotes the probability of transitioning from node j to i and its value is $\frac{1}{\text{Outbound Link Number of Node } j}$, as illustrated in Figure 2.1.

According to our intuition, the PageRank score of node i depends on the PageRank scores of the nodes with edges pointing to node i . Therefore, the PageRank score of node i can be computed as:

$$PR_i = \sum_{\forall j, j \rightarrow i} \frac{PR_j}{\text{Outbound Link Num of Node } j} \quad (2.1)$$

In order to make the equation more compact, we use PageRank vector $\vec{x}$ to present the PageRank score for each node and $\vec{x}$ is the solution of the eigenvalue equation:

$$\vec{x} = \mathbf{P} \cdot \vec{x} \quad (2.2)$$

In some cases, a node may have no outbound links and its PageRank score cannot be distributed to others. Considering these special nodes, an additional teleportation vector $\vec{v}$ weighted by the damping parameter d is attached to Equation(2.2):

$$\vec{x} = d \cdot \mathbf{P} \vec{x} + (1 - d) \cdot \vec{v} \quad (2.3)$$

where $\vec{v}$ is a positive vector and $\sum v_i$ is 1. When the network scale grows, it is harder to find the exact solution for the above equation in a reasonable time. Therefore, Page et al. [112] introduced an iterative approach to get the approximate solution. The equation for the k^{th} iteration is defined as:

$$\vec{x}^{(k)} = d \cdot \mathbf{P}\vec{x}^{(k-1)} + (1 - d) \cdot \vec{v} \quad (2.4)$$

and the initial value of $\vec{x}$ can be set as $\vec{v}$ or $\vec{0}$. For the example in Figure 2.1, if we use damping coefficient $d = 0.85$, vector $\vec{v} = [\frac{1}{n}, \frac{1}{n}, \dots, \frac{1}{n}]^T$, and the initial PageRank vector $\vec{x}^{(0)} = \vec{v}$, after 25 iterations, the PageRank scores of nodes A , B , C and D are 0.3134, 0.2278, 0.1343 and 0.3246, respectively. These scores indicate the importance of each node. Recall that D is pointed by all others and becomes the most important node. A is the only node pointed by D , hence it is the second important node in the network. A and C together have two outbound links pointing to B , whose importance is lower than A . C 's score is the lowest since it only has one inbound link from B .

Not only can PageRank rank web pages, but it also has been widely applied to various other domains. Gleich [55] surveyed the diversity of applications of PageRank and concluded that PageRank can be applied to Chemistry, Biology and Bioinformatics, Neuroscience, Bibliometrics, Databases and Knowledge Information Systems, Recommender Systems, Social Networks Web, i.e., twelve domains in total. Recently, PageRank-based techniques have also been proposed to analyze software systems. Chepelianskii [39] used PageRank to analyze function importance for Linux kernel. Kim et al. [77] proposed

MonitorRank, a PageRank-based approach to find the root causes of anomalies in service-oriented architectures. Bhattacharya et al. [28] proposed the notion of NodeRank based on PageRank to measure the importance of nodes on a static graph for software analysis and fault prediction. Later on, Mirshokraie et al. [103] proposed the notion of FunctionRank, a dynamic variant of PageRank, for ranking functions in terms of their relative importance, for mutation testing.

2.1.2.2 HITS and SALSA

HITS is proposed by J.Kleinberg [81]. Different from PageRank, HITS does not only use the connections between webpages, but it also considers the correlation between webpage and the topic which users provide for query. This insight implies a webpage plays two roles in the context of the query. First, a webpage can be an *authority* if it can provide useful information about the topic. Second, the webpage can be a *hub* if it can provide more links to good authorities on the topic. The HITS algorithm is designed to iteratively compute the authority and hub score of each webpage.

Specifically, HITS first applies the traditional text matching technique to filter out a bunch of webpages which are irrelevant to the topic, and it gets a directed and topic-related graph G . In the next step, HITS constructs an adjacent matrix A based on G , and the iterative functions of HITS can be presented as follows:

$$\vec{a}^{(k)} = \phi(A^T \vec{h}^{(k-1)}), \quad \vec{h}^{(k)} = \phi(A \vec{a}^{(k)}) \quad (2.5)$$

where $\vec{a}^{(k)}$ and $\vec{h}^{(k)}$ denote the vector of authority and hub scores in the k^{th} iteration, and $\phi(\vec{x})$ is the normalization function that $\phi(\vec{x}) = \frac{\vec{x}}{\|\vec{x}\|_2}$. Typically, the initial values of $\vec{a}$ and $\vec{h}$ can be selected as:

$$\vec{a}^{(0)} = \vec{h}^{(0)} = [\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}, \dots, \frac{1}{\sqrt{n}}]^T \quad (2.6)$$

where n is the number of nodes of G .

SALSA (Stochastic Approach for Link Structure Analysis) is proposed by Lempel and Moran [90], which integrates the authority and hub idea from HITS, and the random walk idea from PageRank. Similar to HITS, SALSA also computes the authority and hub score for each webpage and the iterative functions of SALSA are updated as:

$$\vec{a}^{(k)} = W_c^T W_r \vec{a}^{(k-1)}, \quad \vec{h}^{(k)} = W_r W_c^T \vec{h}^{(k-1)} \quad (2.7)$$

where W_c and W_r are the matrices generated from A by dividing each entry of A by the sums of its column and row, respectively [49].

In this work, we integrate SBFL with PageRank algorithm to boost SBFL since (1) SBFL does not utilize any text or code similarity between tests and methods for fault localization, and PageRank algorithm is query-independent as well; (2) HITS and SALSA introduce two attributes (authority and hub) of each node, but PageRank only has one attribute (importance) of node, which is more compact than HITS and SALSA. To the best of our knowledge, this work is the first to apply the PageRank algorithm for fault localization.

2.1.3 Method-level Aggregation

Method-level Aggregation (MA) is a technique developed by Sohn et al. [142]. Instead of using method coverage to calculate SBFL scores, MA uses coverage of all the associated statements within a method and calculates the corresponding SBFL scores for all of them. Next, for each method, the statement-level scores are ranked and the highest SBFL score is identified as the SBFL score of the associated method. MA has been demonstrated to be advanced as follows:

- **Applicability:** Method-level Aggregation is to aggregate statement coverage into method level, and thus can be applied to any SBFL technique.
- **Effectiveness:** In previous work [142], empirical evaluation of this technique was applied to existing SBFL formulas, which shows the formulas with Method-level Aggregation can rank about 42% more faults at the top.

Figure 2.2 shows an example of code snippet to illustrate the benefits of MA. In this code snippet, `m1` is faulty, because `m1` should be a parity-check code, the output `s` will return 1 if the input is an even number, and return -1 otherwise. We assume that there are 4 tests (`t1-t4`), and `t1` is failing. Therefore the spectrum tuple (e_f, e_p, n_f, n_p) of both `m1` and `m2` is $(1, 2, 0, 1)$, resulting in $Ochiai = \frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}} = \frac{1}{\sqrt{1(1+2)}} = 0.578$ and $OP2 = e_f - \frac{e_p}{e_p+n_p+1} = 1 - \frac{2}{2+1+1} = 0.50$. The code defect exists in `m1`, so `m1` is expected

```

1 class Code{
2   static int m1(int x) {
3     int y = Math.abs(x);
4     if ( y % 2 == 1)
5       int s = 1; //buggy
6     else
7       int s = 1;
8     return s;
9   }
10
11
12   static int m2(int x) {
13     int s = x + 1;
14     return s;
15   }
16 }

1 public void t1() {
2   int a = Code.m1(-2);
3   int b = Code.m2(a);
4   assertEquals(2, b);
5 }
6 public void t2() {
7   int a = Code.m1(2);
8   assertEquals(1, a);
9 }
10 public void t3() {
11   int a = Code.m1(3);
12   int b = Code.m2(a);
13   assertEquals(0, b);
14 }
15 public void t4() {
16   int a = Code.m2(5);
17   assertEquals(6, a);
18 }

```

Figure 2.2: Example Code Snippet for Method-level Aggregation
to have a higher SBFL score than m2. However, since m1 and m2 share the same spectrum, the traditional SBFL techniques assign the same score for both methods and make it impossible to differ m1 from m2 regarding their suspiciousness.

In order to improve SBFL in the scenarios similar to Figure 2.2, we adapt MA to select the statement with the highest SBFL score within each method. In general, MA utilizes the fine-grained statement-level information to help with precise method-level fault localization [142]. When applying MA to the example, we use the traditional SBFL techniques (Ochiai and Op2) to assign SBFL scores for all statements in m1 and m2, and the faulty statement has the highest SBFL score (the Ochiai and Op2 scores are both 1.0). Then,

according to the fine-grained statement-level analysis, the SBFL score of `m1` is now higher than that of `m2` due to the high SBFL scores for the statements within `m1`.

Note that PRFL only focuses on method-level fault localization, making it impossible to directly apply MA after PRFL due to the lack of statement-level fault localization results. Therefore, in this work, instead of directly using the highest statement SBFL score as the SBFL score of the associated method, MA is adapted such that for each method the spectrum of the statement with the highest SBFL score substitutes the method's spectrum. For instance, (e_f, e_p, n_f, n_p) of `m1` becomes $(1, 0, 0, 3)$ which is the same as the faulty statement and correctly reflects the suspiciousness of program methods. In this way, the adjusted method spectrum is used for more advanced PRFL.

2.2 Motivating Example

SBFL is designed based on program execution statistics, which include both test coverage and test outcomes. Execution statistics can be treated as the information source of SBFL’s analysis so that they determine the upper bound of SBFL’s accuracy. When execution statistics are built, SBFL would distribute them to construct program spectra, then apply various ranking formulas to compute suspiciousness score for each program entity. Program spectrum is a practical way to present execution statistics, however, it is risky since it loses useful information during construction. Here an example will be analyzed to show how program spectra affect the accuracy of fault localization.

It can be shown from Figure 2.3 that in example class `Code`, method `m2` is faulty since its conditional expression should be $(x > 10)$ instead of $(x > 1)$. This fault leads to `t1` and `t2` to fail. Based on the spectrum information in the left half of Table 2.2, the traditional Tarantula technique would compute all the suspiciousness scores of `m1`, `m2`, and `m3` as the same, i.e., 0.5. This result is no better than a random guess, and thus it is not quite helpful for fault localization. However, when we observe the detailed test coverage shown in Figure 2.4, we can directly find that `m2` is faulty since `t2` fails and only covers `m2`. This example shows that different tests have different capabilities to locate faults, and one limitation of the original spectrum information is that it only focuses on computing how many failing and passing tests cover the program entities but ignores the test differences. This observation inspires us that if tests can be weighted based on their capabilities in localizing potential faults,

```

1 class Code{
2   static int m1(int x){
3     if (x >= 0)
4       return x;
5     else
6       return -x;
7   }
8   static int m2(int x){
9     if (x > 1)//buggy
10      return x;
11    else
12      return 0;
13  }
14  static int m3(int x){
15    return x * x;
16  }
17}

1 public void t1() {
2   int a = Code.m1(-2);
3   int b = Code.m2(a);
4   int c = Code.m3(b);
5   assertEquals(0, c);
6 }
7 public void t2() {
8   int a = Code.m2(5);
9   assertEquals(0, a);
10}
11 public void t3() {
12   int a = Code.m2(15);
13   int b = Code.m3(a);
14   int c = Code.m1(b);
15   assertEquals(225, c);
16}
17 public void t4() {
18   int a = Code.m2(30);
19   assertEquals(30, a);
20}

```

Figure 2.3: Example Code and Corresponding Test Suite

the SBFL will be more accurate.

Here we just analyze failing tests. By our intuition, the test weight should be impacted by the test capability and the covered program entities. Firstly, if the failing test covers very few program entities, this test has a very small scope and it is more capable to infer faulty entities. Therefore its weight should be high. On the other side, if its covered entities are more likely to be faulty, it in turn also should get a higher weight. Similarly, if a program entity is covered by more highly weighted tests, it should also be more likely to contain faults.

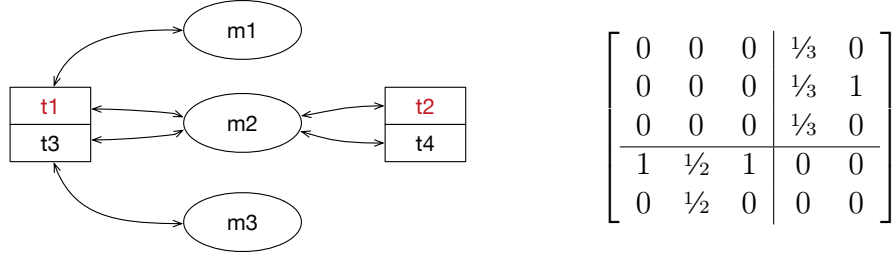


Figure 2.4: Test Coverage Graph of Code and the Transition Matrix of Failing Tests. (Red means the tests that failed.)

All the analysis above is constructed on the bi-directional test coverage graph, which is a kind of network. Hence test weight analysis can be solved by PageRank. Note that the failing and passing tests cover a different set of entities, PageRank analysis will be executed twice to generate the scores of entity importance for failing and passing tests. We term these scores *faultiness* and *successfulness* scores, respectively. For example, when computing the failing test weights, we use vector $\vec{x} = [m_1, m_2, m_3, t_1, t_2]^T$ to present the node values where m_1, m_2 and m_3 present the faultiness scores of m1, m2 and m3 and t_1, t_2 show the test weights of t1 and t2. The test capabilities can be presented by teleportation vector $\vec{v} = [0, 0, 0, w_1, w_2]^T$, where the first three 0s denote the three corresponding source methods and w_1 and w_2 are the weights of t1 and t2. They can be computed by $w_i = \frac{c_i^{-1}}{\sum c_j^{-1}}$, where c_i is defined as the number of program entities covered by the i th test. For this example, c_1 and c_2 are 3 and 1 respectively and $\vec{v}$ is $[0, 0, 0, 0.25, 0.75]^T$. The construction of the transition matrix has been introduced in Section 2.1.2 and the matrix $\mathbf{P}$ can be found in Figure 2.4. Assume that the damping factor d is 0.7, $\vec{x}^{(0)}$ is $\vec{0}$, based on Equation(2.4), we can get $\vec{x}$ as $[0.061, 0.290, 0.061, 0.262, 0.326]^T$, where m_2 is

greater than m_1 and m_3 , indicating that **m2** is highly connected with failed tests and thus is more likely to be faulty, while w_1 is less than w_2 , indicating that **t2** is more effective to help with fault localization. This result reflects that PageRank analysis computes faultiness score for each method and distributes weights for tests in tandem. In the next step, we only utilize faultiness scores to construct weighted spectra since they already include the information from test weights. The weighted spectra can be computed by the normalized faultiness score $\hat{m}_i = \frac{m_i}{\max(\{m_j\})}$. In this example, only failing tests are considered, so only e_{fi} and n_{fi} need to be updated as $\hat{e}_{fi} = \hat{m}_i \cdot N_f$ and $\hat{n}_{fi} = N_f - \hat{e}_{fi}$, where N_f is the total number of failing tests. The passing tests can be analyzed in a similar way, whose details can be found in Section 2.3.1.2. The right half of Table 2.2 shows the weighted spectrum information and updated Tarantula scores for each **Code** method. According to the table, PRFL boosts Tarantula to rank **m2** as the first, demonstrating the effectiveness of PageRank for fault localization.

Although PRFL can help with Tarantula with the above example, some other formulas, e.g., Ochiai, actually can also rank the faulty method precisely. Therefore, we further show another example. Suppose method **m1** is also faulty by changing Line 3 to `if (x ≥ 5)`, and also **t4** is modified as:

```

1          public void t4() {
2              int a = Code.m1(4) + 5;
3              int b = Code.m3(a);
4              assertEquals(81, b);
5          }
```

Program Entity	Original Spectrum Info					Weighted Spectrum Info				
	e_f	e_p	n_f	n_p	T	e_f	e_p	n_f	n_p	T
m1	1	1	1	1	0.5	0.42	1	1.58	1	0.30
m2	2	2	0	0	0.5	2	2	0	0	0.50
m3	1	1	1	1	0.5	0.42	1	1.58	1	0.30

Table 2.2: Original and Weighted Spectra of Code. (T denotes Tarantula score)

Since now both **m1** and **m2** are faulty, tests **t1**, **t2** and **t4** fail and only **t3** pass. This result leads all three methods to share the same traditional spectrum information, i.e., $(e_f, e_p, n_f, n_p) = (2, 1, 1, 0)$ – no matter which SBFL formula is applied, all the methods will be ranked with the same suspiciousness. However, as we analyzed before, **m2** is clearly a fault since it is the only method covered by the failed **t2**. Based on our PRFL idea, we can get vector $\vec{x}$ as $[m_1, m_2, m_3, t_1, t_2, t_4]^T = [0.099, 0.213, 0.099, 0.198, 0.238, 0.151]^T$. This outcome shows that firstly, **m2** is more likely to be faulty than **m1** and **m3** since 0.213 is greater 0.099. Secondly, as we expected, **m1** and **m3** have the same score(0.099) and the reason is that both of them are covered by the same tests. Thirdly, **t2** is the most important (e.g., with the highest weight 0.238) test among **t1**, **t2** and **t4** since it only covers **m2**. Moreover, **t1** is more important than **t4** since it covers more faulty methods. This multiple-fault example further demonstrates the effectiveness of PageRank for fault localization.

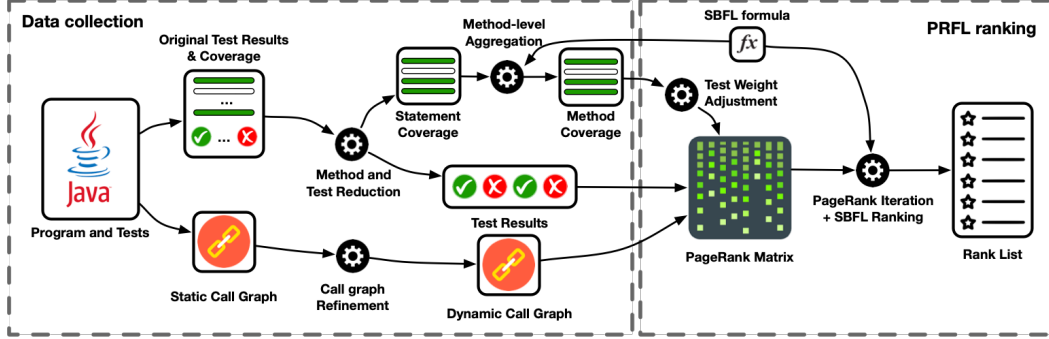


Figure 2.5: Framework of the Proposed Approach

2.3 Approach

In this section, we illustrate the approach of our study. An overall framework of our approach is depicted in Figure 2.5. Specifically, in Section 2.3.1, we propose our basic approach, PRFL, which consists of three major phases: *preparation*, *PageRank analysis*, and *ranking*. In the *preparation* phase (Section 2.3.1.1), PRFL first collects execution traces and test results from the faulty program by running tests and further applies static analysis to construct the connections between program methods, i.e., the call graph. In the *PageRank analysis* phase (Section 2.3.1.2), PRFL constructs the connection matrix between tests and methods as well as among source methods and generates the weighted spectrum information for each test. In the final *ranking* phase (Section 2.3.1.3), PRFL takes as input the updated weighted spectra and uses existing ranking formulas to localize faulty methods.

In Section 2.3.2, we propose PRFL+, which further enhances PRFL on three major parts: *call graph refinement*, *passing test weight adjustment* as well as *test and method reduction*. Considering that not all the methods in static

call graphs are executed during runtime, in the *call graph refinement* part (Section 2.3.2.1), we use dynamic rather than static call graphs to collect the runtime connections of methods. In the *passing test weight adjustment* part (Section 2.3.2.2), we take the differences in the capabilities of passing tests into consideration by adjusting their weights before PageRank analysis. In the *test and method reduction* part (Section 2.3.2.3), since some methods and tests make negligible impacts on the PageRank analysis, we propose two approaches to identify and remove such methods and tests to make PRFL more efficient.

Furthermore, in Section 2.3.3, we propose an aggregation and PageRank-based fault localization technique, namely PRFL_{MA} , which adapts Method-level Aggregation (MA) [142] to utilize the detailed statement coverage information in order to boost PRFL. Specifically, instead of applying SBFL scores directly, PRFL_{MA} firstly identifies the statement with the highest suspiciousness score within each method, then uses its coverage as the method coverage for PRFL. Finally, in Section 2.3.4, we analyze the time and space complexity of PRFL theoretically.

2.3.1 Basic Approach: PRFL

2.3.1.1 Preparation Phase: Static and Dynamic Analysis

The *preparation* phase in PRFL is designed to collect graph data from both dynamic execution and source code for the PageRank analysis. In this phase, PRFL constructs test coverage graph via dynamic analysis. When each test is running, the dynamic analysis performs code instrumentation to auto-

matically record which methods are executed by the test. Note that different methods may be covered by the same failing and passing tests, which means they cannot be differentiated only by the coverage graph; hence, more information should be mined to alleviate this issue. Our heuristic is that if two methods have the same test coverage (e.g., the same method and test connections), the connections between the tied methods and other methods can help break the tie (e.g., a method connected with more fault-prone methods may also be fault-prone since it may have propagated the error states to the connected methods). Therefore, for each method, PRFL further applies static analysis to extract the static call graph to obtain the connections among source methods. The constructed call graph will then be combined with test coverage for the PageRank analysis.

2.3.1.2 Analysis Phase: PageRank Propagation

Here we first recall the original PageRank equation (Equation(2.3)). The PageRank equation consists of three elements, i.e., transition matrix $\mathbf{P}$, damping factor d , and teleportation vector $\vec{v}$. To be specific, damping factor d and teleportation vector $\vec{v}$ are parameters and transition matrix $\mathbf{P}$ is constructed based on the coverage and call graph.

Transition Matrix Construction

As we have illustrated in Section 2.2, given the test coverage information (i.e., the connections between tests and methods), the transition matrices

can be partitioned as:

$$\mathbf{P} = \left[\begin{array}{c|c} \mathbf{0} & \mathbf{P}_{TM} \\ \hline \mathbf{P}_{MT} & \mathbf{0} \end{array} \right] \quad (2.8)$$

where $\mathbf{P}_{MT}$ and $\mathbf{P}_{TM}$ denote the transition matrix between methods and tests (based on the test coverage graph). Note that since the test coverage graph is bipartite, the sub-matrices on the top left and bottom right are zero-matrices.

The matrix $\mathbf{P}$ can present the differences between methods based on test coverage; however, some methods may be covered by the same failing and passing tests and they cannot be differentiated only by $\mathbf{P}$. To alleviate this issue, we utilize the Class Hierarchy Analysis (CHA) call graph algorithm [45] to further distinguish these methods. The transition matrix of the call graph is constructed differently. Firstly, we convert the call graph to adjacent matrix $\mathbf{A}$ as follows: suppose method m_i invokes method m_j , one edge would be added from m_i to m_j with weight 1. Also, m_j will return to m_i when finished and this return relation should also be considered. Heuristically, the calling edges may be more important than the return edges, thus we assign a smaller weight, δ , on the return edge from m_j to m_i , i.e., A_{ij} and A_{ji} can be computed as $A_{ij} = 1$ and $A_{ji} = \delta$. Note that when m_j and m_i invoke each other (e.g., due to recursion), both A_{ij} and A_{ji} are equal to $1 + \delta$. Finally, $\mathbf{A}$ will be column-normalized to $\hat{\mathbf{A}}$ by:

$$\hat{A}_{ij} = \frac{A_{ij}}{\sum_k A_{ik}} \quad (2.9)$$

and the method-to-method matrix (i.e., the transition matrix of call graph), $\mathbf{P}_{MM}$, can be computed as $\mathbf{P}_{MM} = \hat{\mathbf{A}}^T$.

Teleportation Vector Design

As we have discussed before, test weight is not only impacted by the faultiness or successfulness likelihood of covered methods, but also by test capabilities. Hence, PRFL uses different teleportation vectors to present the impact of test capabilities. Teleportation vector $\vec{v}$ is composed of two sub-vectors: $\vec{v} = [\vec{v}_m^T, \vec{v}_t^T]^T$, where $\vec{v}_m$ and $\vec{v}_t$ denote the teleportation vector of methods and tests respectively. For both failing and passing tests, $\vec{v}_m$ is $\vec{0}$. For failing tests, $\vec{v}_t$ is $[w_1, w_2, \dots, w_m]^T$, where $w_i = \frac{c_i^{-1}}{\sum c_j^{-1}}$ and c_i denotes the number of methods covered by test i . This setting is based on the property that if a test fails, it covers at least one faulty method so that a failing test with a smaller scope is more capable to locate the faults. However, a passing test does not satisfy this property since it may cover a faulty method whose fault is not triggered. Therefore, the capability of a passing test does not depend on the number of covered methods and all test capabilities share the same weight. Due to the normalization of weight w_i , $\vec{v}_t$ for passing tests is defined as $[\frac{1}{t}, \frac{1}{t}, \dots, \frac{1}{t}]^T$, where t denotes the number of the passing tests.

Constrained PageRank Algorithm

In this work, we extend the standard PageRank algorithm [112] to analyze the integration of test coverage and call graph. The standard PageRank algorithm is applied to a graph whose nodes and edges are all congeneric. However, in our application scenario, this prerequisite is not satisfied since the edges in the test coverage graph present the connections between methods and

tests, which are not congeneric with the ones in call graph. Thus, the standard PageRank algorithm may be underperformed because of edge variance.

To overcome this issue, we propose a constrained PageRank algorithm to differentiate the edges based on their connection types. Our intuition is that the test coverage graph makes the main contribution for fault localization, so the edges in test coverage graph deserve more weights; on the other hand, the edges in call graph should be set less weights. We use parameter α to tune the weight of call graph, and the constrained transition matrix is:

$$\mathbf{P} = \left[\begin{array}{c|c} \alpha \mathbf{P}_{MM} & \mathbf{P}_{TM} \\ \hline \mathbf{P}_{MT} & \mathbf{0} \end{array} \right] \quad (2.10)$$

The PageRank vector $\vec{x}$ in Equation(2.3) can be decomposed as $\vec{x} = [\vec{x}_m^T, \vec{x}_t^T]^T$, where $\vec{x}_m$ denotes the method faultiness or successfulness scores and $\vec{x}_t$ denotes the weights of tests. Then, the iterative Equation(2.3) can be updated as:

$$\vec{y}_m^{(k)} = d \cdot (\alpha \mathbf{P}_{MM} \cdot \vec{x}_m^{(k)} + \mathbf{P}_{TM} \cdot \vec{x}_t^{(k)}) \quad (2.11)$$

$$\vec{y}_t^{(k)} = d \cdot \mathbf{P}_{MT} \cdot \vec{x}_m^{(k)} + (1 - d) \cdot \vec{v}_t \quad (2.12)$$

$$\vec{x}_m^{(k+1)} = \frac{\vec{y}_m^{(k)}}{\max(\vec{y}_m^{(k)})} \quad (2.13)$$

$$\vec{x}_t^{(k+1)} = \frac{\vec{y}_t^{(k)}}{\max(\vec{y}_t^{(k)})} \quad (2.14)$$

where both initial settings of $\vec{x}_m$ and $\vec{x}_t$ are $\vec{0}$.

2.3.1.3 Ranking Phase: Weighted Spectrum-Based Fault Localization

The weighted spectrum can be constructed using faultiness, successfulness score and failing, passing test number. In the second phase, the *PageRank Analysis* is executed twice to generate vector $\vec{x}_{mf}$ and $\vec{x}_{ms}$, which consist of faultiness and successfulness scores of all covered methods. For each method m_i , its faultiness score s_{fi} and successfulness score s_{si} are extracted from $\vec{x}_{mf}$ and $\vec{x}_{ms}$ respectively, and the weighted spectrum can be computed as:

$$\begin{aligned}\hat{e}_{fi} &= s_{fi} \cdot N_f, & \hat{n}_{fi} &= N_f - \hat{e}_{fi} \\ \hat{e}_{pi} &= s_{si} \cdot N_p, & \hat{n}_{pi} &= N_p - \hat{e}_{pi}\end{aligned}\tag{2.15}$$

where N_f and N_p denote the total number of failing and passing tests, respectively. PRFL then applies SBFL formulas to compute suspiciousness scores on weighted spectra and ranks all method. The weighted spectra include information not only from test coverage, but also from test capabilities and call graph information. Therefore, the weighted spectra are more accurate to reflect the method faultiness and successfulness, and can consequently boost the effectiveness of SBFL techniques.

2.3.2 Extended Approach 1: PRFL+

2.3.2.1 PRFL+ Call Graph Refinement

Generally, the call graph represents the method invocation relations in a given program, which helps developers efficiently understand the code structure. In the literature, various static analysis techniques [45, 64, 108] have

been proposed to capture static call graphs for Java programs to obtain the connections among source code methods.

In our previous work [170], we analyzed source code and directly applied static call graph analysis to build the connections of program entities (i.e. methods). However, this setting could count more connections than necessary since some methods in static call graphs might not be executed during test execution. In this work, we explore the efficacy of PRFL by replacing static call graphs with dynamic call graphs traced during runtime for PageRank analysis to study the impact of call graph types.

2.3.2.2 PRFL+ Passing Test Weight Adjustment

The original PRFL only differentiates failing tests by the number of their covered methods to explore the effects on fault localization, while the effects of differentiated passing tests are not yet to be studied. In particular, recall that in section 2.3.1.2, we design the teleportation vectors to present the impact of test capabilities in PRFL. Specifically, the teleportation vector $\vec{v} = [\vec{v}_m^T \ \vec{v}_t^T]^T$, where $\vec{v}_m$ and $\vec{v}_t$ represent the teleportation vectors of methods and tests respectively. For passing tests, $\vec{v}_m$ is designed to be $\vec{0}$, and the weight of $\vec{v}_t$ is defined as $[\frac{1}{t}, \frac{1}{t}, \dots, \frac{1}{t}]^T$ after normalization, where t denotes the number of passing tests. As a result, all passing test capabilities are assigned with the same teleportation weights based on this assumption that the capability of a passing test is not influenced by the number of its covered methods.

In this section, we study the effects of adjusting the teleportation

weights $\vec{v}_t$ for the passing tests. In Section 2.3.1.2, we design the teleportation weights of failing tests based on the property that a failing test covering very few program entities will have a smaller scope, which is more helpful to infer faulty entities. Here, we make a similar assumption that the teleportation weights of passing tests should follow the same principle: if the passing test covers very few program entities, these entities would more likely be fault-free, and this indicates the passing test is more important and should be assigned a higher weight. The reason is that such passing tests would have failed with high probability if the element is faulty since it is easy for any fault within the covered element to propagate to the test output due to the small number of covered elements. Therefore, we design the teleportation weights of passing tests $\vec{v}_t$ as $[w_1, w_2, \dots, w_m]^T$, where $w_i = \frac{c_i^{-1}}{\sum c_j^{-1}}$ and c_i denotes the number of methods covered by test i , which are the same as failing tests.

2.3.2.3 PRFL+ Test and Method Reduction

Recall that in the *PageRank analysis* phase, the faultiness and successfulness scores of all suspicious methods (i.e., the methods covered by failing test cases) and their spectra are computed based on the construction of the method-to-test transition matrix $\mathbf{P}_{MT}$ and test-to-method transition matrix $\mathbf{P}_{TM}$, whose computational costs strongly depend on the matrix dimensions. Furthermore, note that not all tests and methods contribute to computing their corresponding teleportation vectors and thus impact negligibly on modeling PRFL. These tests and methods are listed as follows:

- Methods: which are covered only by passing tests
- Tests: which cover the methods that are not covered by any failing tests

Accordingly, we propose two approaches: test-based reduction and method-based reduction, to identify such tests and methods and remove them prior to running PRFL for the purpose of reducing computational cost.

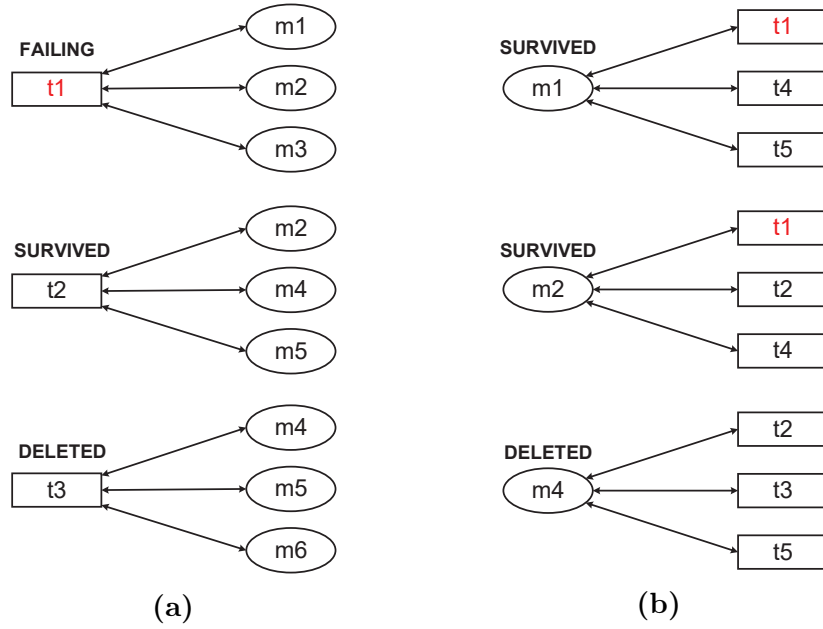


Figure 2.6: Test-based Reduction (a) and Method-based Reduction (b)

Test-Based Reduction

The test-based reduction approach removes the elements in P_{TM} and P_{MT} when the corresponding tests contain no failing-test-covered methods. For instance, in Figure 2.6 (a), t2 and t3 are passing tests. t1 is a failing test (in red) that covers the methods m1, m2, and m3. Since the passing test

$\mathbf{t2}$ also covers $\mathbf{m2}$ which is covered by $\mathbf{t1}$, the test-based reduction approach retains all the $\mathbf{t2}$ -related elements ($\mathbf{t2-m2}$, $\mathbf{t2-m4}$, $\mathbf{t2-m5}$). On the other hand, the passing test $\mathbf{t3}$ covers $\mathbf{m4}$, $\mathbf{m5}$, and $\mathbf{m6}$, and none of which is covered by $\mathbf{t1}$. Therefore, all the $\mathbf{t3}$ -related elements ($\mathbf{t3-m4}$, $\mathbf{t3-m5}$, $\mathbf{t3-m6}$) would be deleted from $\mathbf{P}_{TM}$, and dually, $\mathbf{m4-t3}$, $\mathbf{m5-t3}$ and $\mathbf{m6-t3}$ would also be deleted from $\mathbf{P}_{MT}$.

Method-Based Reduction

Similarly, the method-based reduction approach removes the elements in $\mathbf{P}_{MT}$ and $\mathbf{P}_{TM}$ when the corresponding methods are only covered by passing tests. For instance, in Figure 2.6 (b), the test $\mathbf{t1}$ is failed (in red) and the tests $\mathbf{t2-t5}$ are passed. According to the method-based reduction approach, since $\mathbf{m1}$ and $\mathbf{m2}$ are covered by the failing test $\mathbf{t1}$, both of them will be retained for PRFL analysis. On the other side, since $\mathbf{m4}$ is covered by all the passing tests $\mathbf{t2}$, $\mathbf{t3}$, and $\mathbf{t5}$, all the $\mathbf{m4}$ -related elements ($\mathbf{m4-t2}$, $\mathbf{m4-t3}$, $\mathbf{m4-t5}$) will be deleted from $\mathbf{P}_{MT}$, and dually, $\mathbf{t2-m4}$, $\mathbf{t3-m4}$, $\mathbf{t5-m4}$ will also be deleted from $\mathbf{P}_{TM}$.

2.3.3 Extended Approach 2: PRFL_{MA}

In Section 2.1.3, it is observed that the traditional SBFL techniques cannot differentiate the faultiness of $\mathbf{m1}$ and $\mathbf{m2}$ only using method coverage. In order to improve the accuracy of fault localization, we further propose PRFL_{MA}, a novel technique which integrates Method-level Aggregation and

PRFL.

In general, PRFL_{MA} includes the following steps:

- Step 1: PRFL_{MA} extracts the spectrum (e_f, e_p, n_f, n_p) for all statements and applies SBFL formula (e.g., Ochiai2) to compute the suspiciousness scores of all statements.
- Step 2: For each method, PRFL_{MA} chooses the most suspicious statement in the given method, and uses the statement coverage to replace the method coverage.
- Step 3: PRFL_{MA} executes PRFL on the updated coverage to rank all the methods.

C	Subject	t1	t2	t3	t4	Och.2	Agg.
m1	<i>static int m1(int x) {</i>	1	1	1	0	0.08	1*
s1	<i> y = Math.abs(x);</i>	1	1	1	0	0.08	0.08
s2	<i> if(y % 2 == 1)</i>	1	0	0	0	1*	1*
s3	<i> int s = 1; //buggy</i>	1	0	0	0	1*	1*
s4	<i> else</i>	0	1	1	0	0	0
s5	<i> int s = y;</i>	0	1	1	0	0	0
s6	<i> return s; }</i>	1	1	1	0	0.08	0.08
m2	<i>static int m2(int x) {</i>	1	0	1	1	0.08	0.08
s7	<i> int s = x + 1;</i>	1	0	1	1	0.08	0.08
s8	<i> return s; }</i>	1	0	1	1	0.08	0.08
Test case outcome		f	p	p	p	m1/2	m1
Number of candidates		4					

Table 2.3: The Process of PRFL_{MA}

We use Table 2.3 as an example to illustrate the process of PRFL_{MA} . Columns **t1-t4** demonstrate the coverage information of four test cases on **m1**,

m2, and their statements. Specifically, 1 and 0 represent the method/statement being covered and uncovered by each test, respectively. For instance, **m1** is covered by **t1**, **t2**, and **t3**. Also, line **s4** in **m1**, "else{" is covered by **t2** and **t3**. Column **Ochiai2** presents the results of the traditional Ochiai2 scores of the two methods and their statements. It can be shown that **m1** and **m2** are assigned with the same SBFL scores and hence cannot be properly ranked because they have the same spectra. On the other hand, the SBFL scores of statements are different. The reason is that some methods are partially executed by tests, leading to different SBFL scores among the statements. Column **Agg.** shows that the statements **s2** and **s3** are assigned with the highest SBFL score within **m1** after applying Ochiai2. Next, the spectrum and SBFL scores of **s2** or **s3** are assigned to **m1**. In other words, originally, **m1** is executed by the tests **t1-t3**, and **s2** and **s3** are executed by **t1**. After applying MA, **m1** is perceived to be executed by **t1**, leading to a Method-level Aggregation of the statement-level spectra. As a result, **m1** is ranked higher than **m2**.

2.3.4 Computational Complexity

Suppose the numbers of failing and passing tests are T_f and T_p , and the numbers of their covered methods are M_f and M_p respectively. In general, the program quality is relatively high and only a few of tests are failed, hence, we can make a mild assumption that $T_p \gg T_f$ and $M_p \gg M_f$. Therefore, the complexity of our approaches is derived mostly based on the complexity of the passing tests. Recall that in Equation 2.11 and 2.12, vector $\vec{x}_m$ and $\vec{x}_t$ denote

the PageRank scores of methods and tests, and their sizes are M_p and T_p . The dimensions of matrix P_{MM} , P_{TM} and P_{MT} are $M_P \cdot M_P$, $M_P \cdot T_P$ and $T_P \cdot M_P$, respectively. Hence, the time complexity of computing Equation 2.11 and 2.12 depends on the complexity of the multiplication of matrix and vector. Specifically, the time complexity of Equation 2.11 is $O(M_P^2 + M_P \cdot T_P)$ and the time complexity of Equation 2.12 is $O(M_P \cdot T_P)$. Therefore, the total time complexity of PRFL is $O(M_P^2 + M_P \cdot T_P)$. On the other hand, compared to SBFL, we need more space to store these matrices, therefore, the space complexity is $O(M_P^2 + M_P \cdot T_P)$ as well.

2.4 Experimental Setup

Our experimental study aims to answer the following research questions:

- **RQ1:** How does PRFL compare with traditional SBFL techniques in term of effectiveness and efficiency?
- **RQ2:** How do different configurations impact the effectiveness of PRFL?
- **RQ3:** How do different numbers of faults impact the effectiveness of PRFL?
- **RQ4:** How does PRFL compare with the recently proposed learning-based and spectrum-based fault localization technique?
- **RQ5:** How does PRFL compare with the recently proposed mutation-based fault localization techniques?
- **RQ6:** How does PRFL perform using different link analysis algorithms?
- **RQ7:** How does PRFL+ perform on localizing real faults?
- **RQ8:** How does PRFL_{MA} perform on localizing real faults?
- **RQ9:** How do PRFL and PRFL_{MA} perform on a large number of artificial faults?
- **RQ10:** How statistically significant is the improvement of the proposed techniques?

2.4.1 Subjects

Real-Fault Subjects We use Defects4J [73] and Bugs.jar [131] as the benchmarks for our evaluations. Defects4J [73] is a mature real fault dataset for testing experiments, and has been widely used in software testing research [25, 74, 84, 120, 137]. Defects4J(v1.2.0, update in December 2017) includes 395 real faults from 6 open-source projects: JFreeChart, Google Closure Compiler, Apache Commons Lang, Apache Commons Math, Joda-Time and Mockito. For each fault, Defects4J provides the faulty program, the fixed program with minimum code change and the failing tests. We identify the faulty methods in the following ways. Firstly, we compare the modified source files to collect code changes. If all changes are located in a single method, we label such method as a faulty method. However, in some other cases, the program changes are distributed in multiple methods which may not all be faulty. To precisely identify the actual fault-triggering methods, we then manually apply all the possible combinations of the modified methods to get the minimum change set that can pass all tests. Note that we use all the 395 Defects4J faults except the faults not within method bodies. Table 2.4 (except the last row) shows the statistics of the Defects4J subjects – Column 1 presents the subject IDs that will be used in the remaining text; Column 2 presents the full name of the subjects; Column 3 presents the number of faults for each subject; finally, Columns 4 and 5 present the LoC (i.e., Lines of Code) and test numbers of the most recent version of each subject in Defects4J.

ID	Program	#Faults	LoC	#Tests
Chart	JFreeChart	26	96K	2,205
Closure	Closure Compiler	133	90K	7,927
Lang	Commons Lang	65	22K	2,245
Math	Commons Math	106	85K	3,602
Time	Joda-Time	27	28K	4,130
Mockito	Mockito	38	23k	1,366
Real-Fault Total	6 Projects	395	344k	21,475
Mutation-Fault Total	240 Projects	96925	2565K	19556

Table 2.4: Subject Statistics

Bugs.jar [131] is a recently proposed real fault dataset, which includes 1158 faults and patches extracted from 8 open-source Java projects: Accumulo, Camel, Apache Commons Math, Flink, Jackrabbit Oak, Log4J2 Logging, Maven, Wicket. In our experiments, we successfully built and applied 560 out of 1158 bugs (Accumulo 56/98, Apache Commons Math 89/147, Flink 69/70, Jackrabbit Oak 238/278, Log4J2 Logging 65/81, Wicket 43/289) to evaluate our approach.

Artificial-Fault Dataset. Although Defects4J and Bugs.jar are suitable for evaluating testing techniques, their projects and faults are rather limited, posing threats to validity. Meanwhile, mutation faults have been shown to be suitable for software testing experimentation [24, 74]. Therefore, we further use the PIT mutation testing tool [15] (with all its 16 supported mutation operators) to generate artificial faults for evaluating PRFL. To be specific, we start from the first 3804 most popular Java projects from GitHub [9]. 1855 projects of those were built successfully with Maven and passed all tests. Then, 786 projects were further removed since PIT crashed or could not terminate within our time limit, i.e., 2 hours. Therefore, finally, we have 240 projects

with mutation faults, ranging from 112 to 313016 lines of code. The last row of Table 2.4 presents the statistics of the mutation faults used in our study.

2.4.2 Implementation and Supporting Tools/Platform

Data Preparation. We use ASM bytecode analysis framework [8] together with JavaAgent [10] to perform on-the-fly code instrumentation to capture the test coverage for each test. Furthermore, we also implement the static Class Hierarchy Analysis (CHA) call graph algorithm [45] based on ASM framework. Note that we ignore all the 3rd party libraries and Java internal libraries during the call graph analysis for time efficiency.

Data Analysis. We use Numpy [14], one of the most popular scientific computing package in Python, to implement and evaluate PRFL and other traditional SBFL techniques. PRFL applies an iterative algorithm to compute the faultiness and successfulness scores. For our application scenario, the test number is relative small, which leads the sizes of transition matrices $\mathbf{P}_{TM}$ and $\mathbf{P}_{MT}$ limited. On the other hand, since the method invocations are not frequent, the transition matrix $\mathbf{P}_{MM}$ is sparse. This property makes PRFL execute fast, and for all the following experiments, PRFL is iterated 25 times for both failing and passing tests.

Tool Supports. Currently all the proposed techniques (including PRFL and PRFL_{MA}) have been implemented and integrated into our intelliFL tool². in-

²<http://www.intellifl.org/>

telliFL has been implemented as a practical Maven Plugin for debugging Java programs with JUnit tests, and now is also publicly available in Maven Central Repository [11]. intelliFL currently supports both single-module and multi-module Maven projects, as well as unit (via Maven Surefire Plugin [13]) and integration (via Maven Failsafe Plugin [12]) tests. Furthermore, intelliFL supports tests developed under both JUnit 3 and 4, and can also support source code developed under JDK versions 7 to 9.

Platform. All our experiments were performed on an Intel(R) Xeon E5-4610 v4 CPU(1.80 GHz) with Ubuntu Linux 16.04.

2.4.3 Evaluation Metrics

We use the absolute wasted effort (AWE) and Top-N, two widely used metrics [25, 159, 160] to evaluate the effectiveness of the studied fault localization techniques. Note that all our metrics do not consider test code.

AWE: Given a faulty program and a ranking formula (such as Tarantula), AWE is defined as the ranking number of the faulty method. However, in some cases, there are more than one method sharing the same SBFL score with faulty method, and AWE is defined as the average ranking of all the tied methods. The AWE is computed as:

$$AWE(b) = |\{m | susp(m) > susp(b)\}| + \frac{|\{m | susp(m) = susp(b)\}| + 1}{2} \quad (2.16)$$

where b is the faulty method and m is any candidate method except b and $|\{\cdot\}|$ is the cardinality of a set. The range of AWE is from 1 to the total number

of methods. A smaller AWE means the fault localization is more effective and the ideal value is 1.

Top-N: This metric counts the number of successfully localized faulty methods within the top-N ($N=1, 3, 5$) ranked results. If the faulty methods share the same score, we use the *average* position to present the fault location. Higher Top-N denotes more effective fault localization. Note that this metric can be quite important in practice since developers usually only inspect top-ranked elements, e.g., over 70% developers only check Top-5 ranked elements [82].

2.5 Result Analysis

2.5.1 RQ1: PRFL’s Overall Effectiveness and Efficiency

2.5.1.1 Effectiveness of PRFL

To answer this RQ, we present the experimental results of PRFL using the default configuration ($d=0.7$, $\alpha=0.001$ and $\delta=1.0$) on all the real faults from the Defects4J dataset. Table 2.5 presents the overall results. In the table, different columns present different effectiveness metrics. (for each metric, Column S represents the traditional spectrum-based techniques while Column P represents our PRFL) and different rows present the subjects and fault localization formulas used. Also, the bottom portion of the table presents the overall results for all Defects4J subjects, e.g., the total Top-N values and the average AWE values. From Table 2.5, we can have the following observations. First, overall Ochiai and Dstar2 (marked in gray) are the two most effective SBFL techniques for all the faults in Defects4J. Also, both of them are able to localize 109+ faults within Top-1 and 209+ faults within Top-3. Second, in general, PRFL is able to boost all the studied traditional SBFL techniques. For example, the overall Top-1/3/5 and AWE values of all traditional techniques are all outperformed by the corresponding PRFL techniques. Third, interestingly, PRFL tends to boost more effective traditional SBFL techniques even more. For example, PRFL is able to boost the number of faulty methods ranked as Top-1 by Ochiai from 79 to 110 (i.e., **39% more**), a higher improvement than the other inferior techniques.

Table 2.7 presents the performance of PRFL and SBFL on Bugs.jar

Tech	Top-1		Top-3		Top-5		AWE		
	S	P	S	P	S	P	S	P	Impr.
Chart	Tarantula	7 12	20 21	22 24	14.92	13.63	8.63%		
	SBI	7 11	20 21	22 24	14.92	13.63	8.63%		
	Ochiai	6 11	17 20	19 24	11.83	10.21	13.66%		
	Jaccard	6 11	17 20	20 24	12.12	10.60	12.54%		
	Ochiai2	6 12	17 21	21 24	12.15	11.40	6.17%		
	Kulczynski	6 11	17 20	20 24	12.12	10.60	12.54%		
	Dstar2	5 10	16 21	19 24	13.65	9.83	28.03%		
	Op2	5 7	14 17	16 20	66.19	61.37	7.29%		
Lang	Tarantula	21 27	45 50	57 57	5.81	5.26	9.40%		
	SBI	21 27	45 50	57 57	5.81	5.26	9.40%		
	Ochiai	22 30	44 51	56 58	5.35	4.76	10.94%		
	Jaccard	22 28	44 50	56 57	5.39	4.88	9.42%		
	Ochiai2	21 28	45 50	55 57	5.35	5.33	0.29%		
	Kulczynski	22 28	44 50	56 57	5.39	4.88	9.42%		
	Dstar2	23 30	45 50	55 58	5.33	4.65	12.70%		
	Op2	23 29	45 48	56 56	5.48	4.61	15.99%		
Math	Tarantula	24 32	61 65	73 78	8.89	7.36	17.20%		
	SBI	24 32	61 65	73 78	8.87	7.36	17.07%		
	Ochiai	24 33	60 65	73 83	9.25	6.92	25.23%		
	Jaccard	24 32	61 66	73 79	8.87	7.33	17.39%		
	Ochiai2	24 31	61 67	73 79	8.88	7.35	17.16%		
	Kulczynski	24 32	61 66	73 79	8.87	7.33	17.44%		
	Dstar2	24 33	60 65	73 83	9.40	6.91	26.46%		
	Op2	23 28	54 61	65 76	10.66	8.35	21.64%		
Time	Tarantula	5 7	11 14	16 16	25.37	23.30	8.18%		
	SBI	5 7	11 14	16 16	25.37	23.30	8.18%		
	Ochiai	6 7	11 13	18 18	22.93	19.93	13.09%		
	Jaccard	5 7	9 12	18 17	25.41	23.44	7.73%		
	Ochiai2	5 7	11 14	16 17	25.44	23.33	8.30%		
	Kulczynski	5 7	9 12	18 17	25.41	23.44	7.73%		
	Dstar2	6 7	11 12	12 12	25.22	21.48	14.83%		
	Op2	8 4	12 11	14 13	62.67	60.19	3.96%		
Closure	Tarantula	12 13	26 33	36 45	120.30	87.01	27.68%		
	SBI	12 13	26 33	36 45	120.30	87.02	27.66%		
	Ochiai	13 14	27 34	37 47	119.40	86.29	27.73%		
	Jaccard	13 14	27 34	37 47	119.40	86.29	27.73%		
	Ochiai2	13 13	26 33	37 46	120.01	86.83	27.64%		
	Kulczynski	13 14	27 34	37 47	119.40	86.29	27.73%		
	Dstar2	14 21	28 42	39 59	108.23	74.77	30.91%		
	Op2	17 10	33 32	42 43	119.26	87.76	26.41%		
Mockito	Tarantula	6 7	18 22	25 31	39.51	34.89	11.69%		
	SBI	6 7	18 22	25 31	39.51	34.89	11.69%		
	Ochiai	7 9	18 20	25 32	49.91	36.66	26.55%		
	Jaccard	6 7	18 22	25 30	39.32	34.42	12.45%		
	Ochiai2	6 7	18 22	25 30	39.46	35.24	10.70%		
	Kulczynski	6 7	18 22	25 30	39.32	34.42	12.45%		
	Dstar2	7 8	17 19	23 33	57.96	44.34	23.50%		
	Op2	6 5	14 16	19 23	155.43	154.03	0.91%		
Overall	Tarantula	75 98	181 205	229 251	35.80	28.58	20.18%		
	SBI	75 97	181 205	229 251	35.80	28.58	20.17%		
	Ochiai	79 110	179 212	230 274	34.66	25.70	25.85%		
	Jaccard	76 99	176 204	229 254	35.08	27.83	20.68%		
	Ochiai2	75 98	178 207	227 253	35.21	28.25	19.78%		
	Kulczynski	76 99	176 204	229 254	35.08	27.83	20.68%		
	Dstar2	79 109	177 209	221 269	36.63	27.00	26.30%		
	Op2	82 83	172 185	212 231	69.95	62.72	10.34%		

Column **S** and **P** indicate SBFL and PRFL respectively

Table 2.5: Results of SBFL and PRFL on All Defects4J Faults

Sub	COV	CG	DP	Analysis	Ranking	Total
Chart	35.18	66.71	6.49	1.33	0.01	109.72
Closure	231.73	431.71	888.87	6.61	0.01	1558.94
Lang	23.85	22.26	0.81	0.38	0.01	47.31
Math	268.32	106.21	8.33	1.47	0.01	384.34
Mockito	45.37	38.93	4.30	0.15	0.01	88.76
Time	21.56	25.72	19.39	1.32	0.01	68.00
Avg.	104.33	115.23	154.70	1.87	0.01	376.17

Table 2.6: Fault Localization Overheads

dataset. Similarly, as presented in Table 2.5, we can observe that overall Ochiai and Dstar2 are still the two most effective SBFL techniques for all the faults in Bugs.jar. Specifically, by applying PRFL, Ochiai can locate 47 (30%+), 139 (27.5%+), and 187 (18.4%+) faults, and Dstar2 can locate 48 (26.3%+), 141 (22.6%+), and 190 (15.8%+) faults within Top 1, 3 and 5. However, we find the improvements of AWE on Bugs.jar is not globally consistent with the observation on Defects4J. For example, overall PRFL can improve more than 8% AWE using Ochiai and Dstar2, but it increases around 13% AWE on Math.

2.5.1.2 Efficiency of PRFL

We record the overhead of our PRFL technique. Table 2.6 presents the average overhead results for all versions of each subject from Defects4J. In Table 2.6, each column presents the time (seconds) spent in each phase of PRFL while the last column presents the total overhead; each row presents the overhead for each subject while the last row presents the average results for all the subjects. Column **COV**, Column **CG**, and Column **Analysis** respectively represent coverage collection, call graph construction, and PageRank analysis costs. Column **DP** represents the time of processing data, which includes reading data

Tech	Top-1		Top-3		Top-5		AWE			
	S	P	S	P	S	P	S	P	Impr.	
Accumulo	Tarantula	2	2	16	23	20	24	228.66	232.94	-1.87%
	SBI	2	2	16	18	20	22	228.66	233.07	-1.93%
	Ochiai	3	6	14	28	19	32	230.25	229.68	0.25%
	Jaccard	3	6	15	28	19	33	228.95	231.00	-0.89%
	Ochiai2	3	7	13	26	19	28	228.06	231.70	-1.59%
	Kulczynski	3	6	15	28	19	33	228.95	231.00	-0.89%
	Dstar2	4	7	18	27	21	33	232.41	230.57	0.79%
Op2	5	6	20	23	20	26	245.24	236.45	3.58%	
Math	Tarantula	9	12	29	34	34	35	18.01	19.13	-6.21%
	SBI	9	12	29	34	34	35	18.01	19.13	-6.21%
	Ochiai	11	14	29	34	34	35	18.90	21.46	-13.49%
	Jaccard	11	14	29	34	34	35	18.12	19.11	-5.42%
	Ochiai2	11	13	29	34	34	35	18.01	19.11	-6.08%
	Kulczynski	11	14	29	34	34	35	18.12	19.11	-5.42%
	Dstar2	11	14	29	34	34	35	18.96	21.47	-13.25%
Op2	9	12	19	28	25	30	22.09	28.75	-30.14%	
Flink	Tarantula	3	5	20	20	33	31	62.35	53.06	14.90%
	SBI	3	5	20	20	33	31	62.35	53.06	14.90%
	Ochiai	3	4	17	20	28	34	75.32	58.36	22.51%
	Jaccard	3	6	18	22	25	31	70.64	53.80	23.83%
	Ochiai2	4	6	19	22	27	32	65.15	52.90	18.80%
	Kulczynski	3	6	18	22	25	31	70.64	53.80	23.83%
	Dstar2	3	4	17	19	28	31	82.89	62.37	24.76%
Op2	2	3	15	19	24	28	96.73	79.12	18.20%	
Oak	Tarantula	8	8	21	21	36	36	148.54	141.45	4.77%
	SBI	8	8	21	21	36	36	148.54	141.41	4.80%
	Ochiai	10	10	23	22	38	37	140.66	140.10	0.40%
	Jaccard	10	10	23	22	38	37	149.50	142.65	4.58%
	Ochiai2	10	10	23	22	37	36	148.28	141.26	4.73%
	Kulczynski	10	10	23	22	38	37	149.50	142.69	4.55%
	Dstar2	10	11	24	25	40	39	140.31	140.37	-0.04%
Op2	10	11	25	25	35	35	206.58	206.00	0.28%	
Log4j2	Tarantula	2	2	12	13	20	21	145.11	123.64	14.80%
	SBI	2	2	12	13	20	21	145.11	123.58	14.84%
	Ochiai	3	6	12	17	20	23	154.55	133.10	13.88%
	Jaccard	3	6	12	19	20	24	156.99	132.03	15.90%
	Ochiai2	3	5	12	16	21	24	150.45	124.75	17.08%
	Kulczynski	3	6	12	19	20	24	156.99	132.03	15.90%
	Dstar2	4	5	13	18	20	24	163.12	138.77	14.93%
Op2	3	5	12	18	15	21	202.07	165.82	17.94%	
Wicket	Tarantula	5	5	12	12	17	17	145.07	133.12	8.23%
	SBI	5	5	12	12	17	17	145.07	133.20	8.18%
	Ochiai	6	7	14	18	19	26	249.55	215.63	13.59%
	Jaccard	6	7	14	18	19	26	178.07	144.38	18.92%
	Ochiai2	6	7	15	16	20	24	163.85	170.91	-4.31%
	Kulczynski	6	7	14	18	19	26	178.07	144.38	18.92%
	Dstar2	6	7	14	18	21	28	267.15	230.00	13.91%
Op2	6	6	14	17	19	23	311.37	273.23	12.25%	
Overall	Tarantula	29	34	110	123	160	164	124.62	117.22	5.94%
	SBI	29	34	110	118	160	162	124.62	117.24	5.92%
	Ochiai	36	47	109	139	158	187	144.87	133.05	8.16%
	Jaccard	36	49	111	143	155	186	133.71	120.49	9.89%
	Ochiai2	37	48	111	136	158	179	128.97	123.44	4.29%
	Kulczynski	36	49	111	143	155	186	133.71	120.50	9.88%
	Dstar2	38	48	115	141	164	190	150.81	137.26	8.98%
Op2	35	43	105	130	138	163	180.68	164.90	8.74%	

Column **S** and **P** indicate SBFL and PRFL respectively

Table 2.7: Results of SBFL and PRFL on All Bugs.jar Faults

(statement coverage and call graph) and constructing matrices for PageRank analysis. Based on the table, the PRFL technique is lightweight and can finish within around 6 minutes for the studied subject on average. Furthermore, the most time-consuming phase is DP while the overheads of other PRFL phases (i.e., the call graph and PageRank analysis time) are comparably low, e.g., less than 10 seconds for **CG** and less than 2 seconds for **Analysis**. The reason is that DP is implemented in Python, which is relatively slow and also does not enable multi-threading.

Proj.	F-matrix	P-matrix
Chart	0.037	248.618
Closure	0.075	1616.352
Lang	0.002	112.479
Math	0.005	382.071
Time	0.007	207.31
Mockito	0.023	42.011

Unit: MB

Table 2.8: Average Memory Consumption of PRFL

Furthermore, we investigate the memory consumption of PRFL. In Table 2.8, the column "F-matrix" and "P-matrix" present the allocated memory of PageRank matrices of failing and passing tests respectively. It can be found that these results are consistent with our assumption that there are much more passing tests than failing tests. Moreover, we can observe that the memory overhead is less than 2 Gigabyte which can be easily handled by a commonly-configured personal computer.

2.5.2 RQ2: Configuration Impacts

In this section, we extend our experiments with different configurations to investigate the influence of internal factors of PRFL so as to learn how to make PRFL achieve better performance. Figure 2.7 presents the impacts of different damping factors (i.e., d) on the effectiveness of PRFL using the default $\alpha=0.001$ and $\delta=1.0$. In the figure, the x axis presents various damping factor values, while the y axis presents the AWE improvements of PRFL techniques over the original pure SBFL techniques (different formulas are represented using different lines). From the figure, we have the following observations. First, the damping factor does not impact the PRFL effectiveness much. For example, for all the formulas on all the subjects, the largest improvement difference among different damping factors is only 4%. Second, for the majority of cases, when the damping factor increases, the improvement rates slightly decrease. This observation is as expected. The reason is that when damping factor increases, the test capabilities will be distributed a smaller weight, causing it to make fewer contributions in localizing the faults.

Figure 2.8 shows the impact of the call graph weights (i.e., α) using the default $d=0.7$ and $\delta=1.0$. Similar to Figure 2.7, the x axis presents different call graph weights, the y axis presents the improvement rates of PRFL over pure SBFL techniques (different line represents different formulas). From the figure, we have the following observations. First, on all the subjects, the improvement rates of PRFL dramatically increase at the very beginning, but then slowly increase or even decrease for some formulas. One major reason is that when

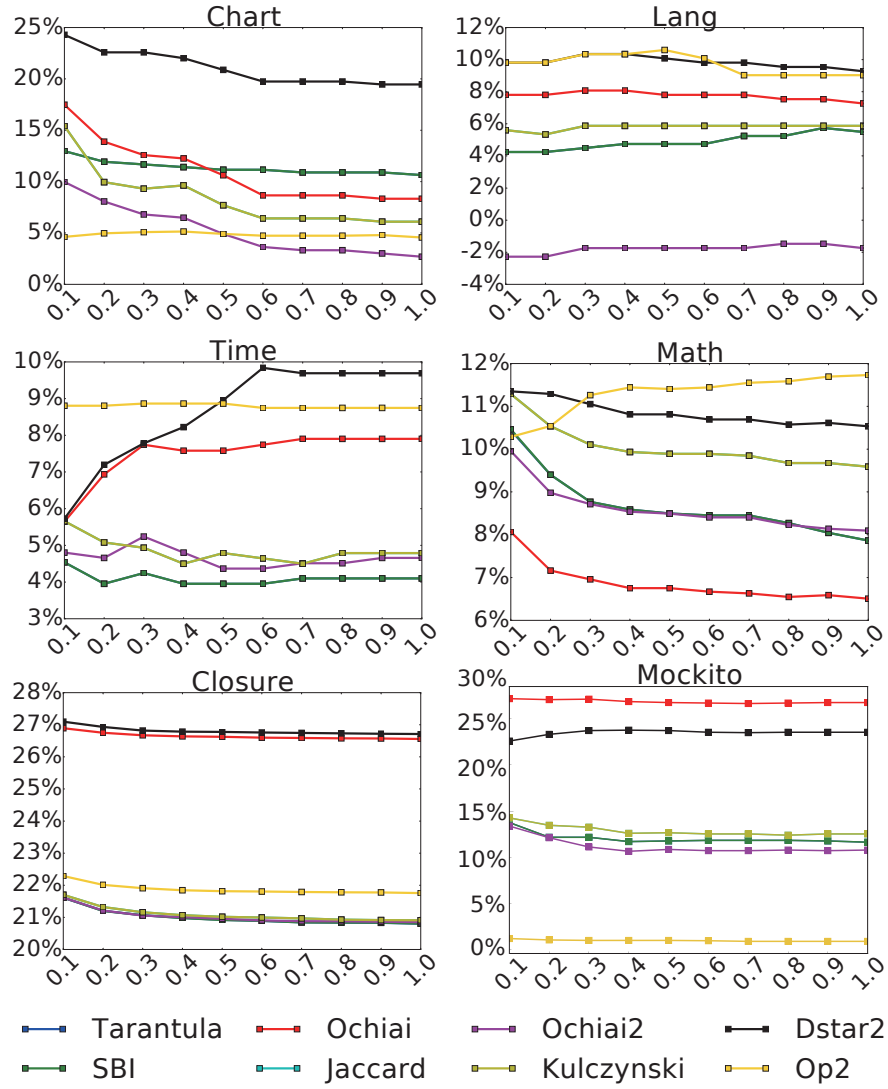


Figure 2.7: Impact of Damping Factor

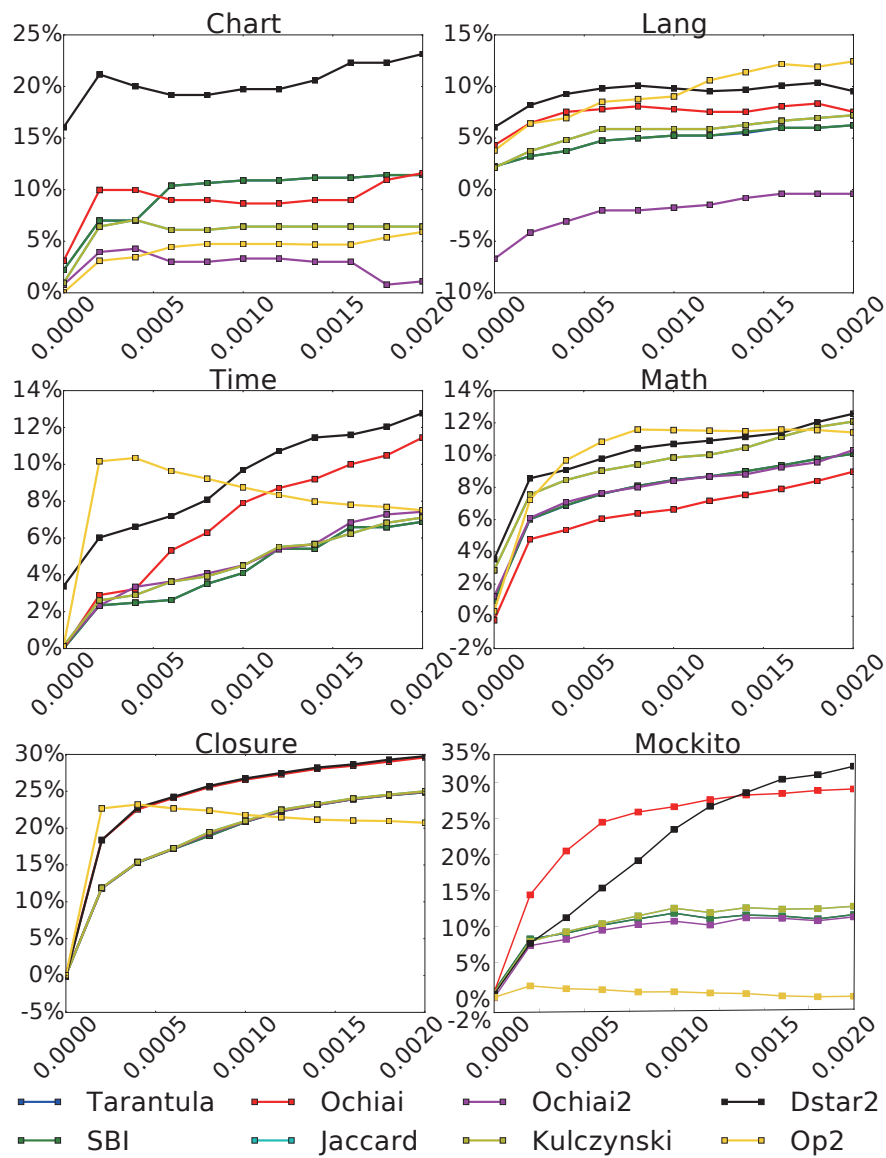


Figure 2.8: Impact of Call Graph Weight

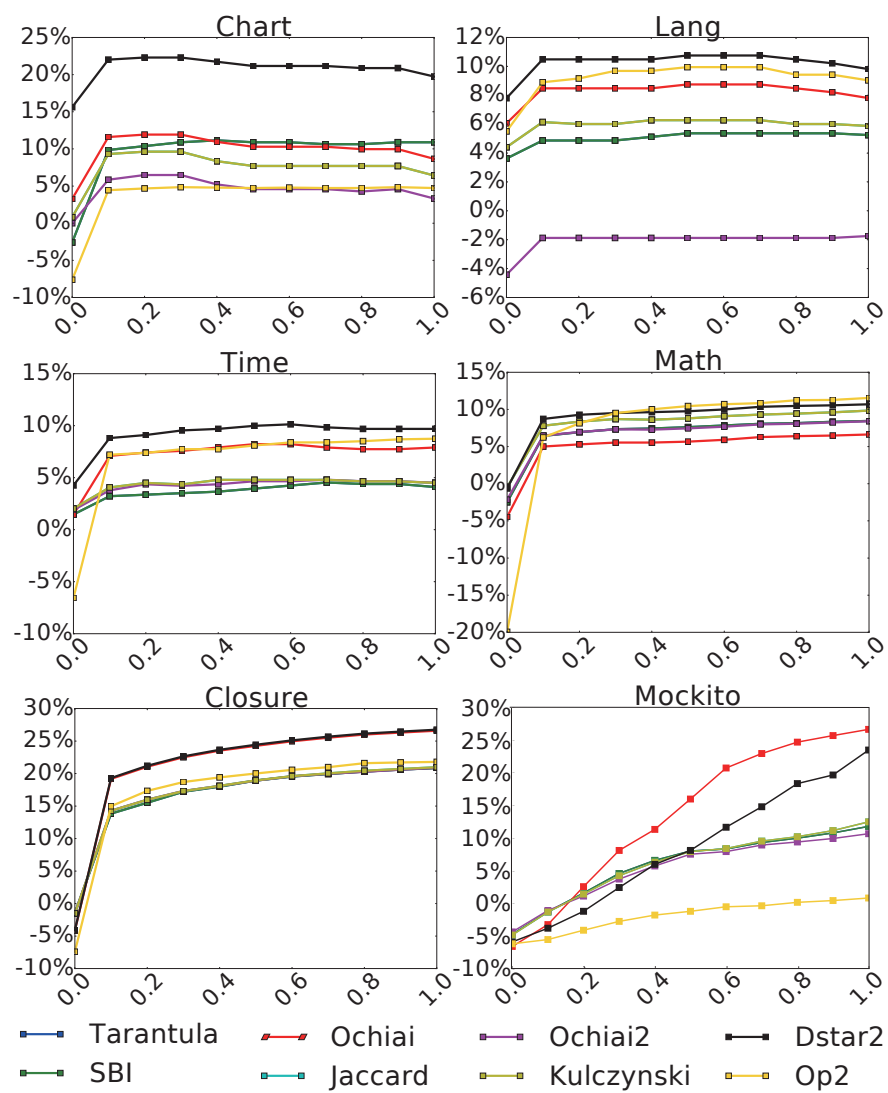


Figure 2.9: Impact of Return Weight

the call graph weight is 0, PRFL degrades to use only the test coverage information and cannot differentiate the methods with the same test coverage, hence the improvement rates are relatively low. Note that even without call graph information (i.e., $\alpha=0$), PRFL is still able to outperform the traditional SBFL techniques for the majority cases. Second, the call graph weight has different impacts on different formulas. For example, the improvement rate of PRFL over Op2 eventually decreases dramatically for subjects Time and Closure, while the improvement rates keep stable or increasing for the other formulas on most subjects. Also, the impact of call graph weight is similar for Tarantula and SBI, as well as Jaccard and Kulczynski due to the similarities in the formula definition.

Figure 2.9 shows the impact of the return edge weights (i.e., δ) using $d=0.7$ and $\alpha=0.001$. Similar to Figure 2.7 and 2.8, the x axis presents different return edge weights, while the y axis presents the improvement rates of PRFL over the traditional SBFL techniques. From this figure, it is observed that the improvement rates of PRFL (especially on Op2) on all the subjects are low and sometimes even below zero when $\delta=0.0$. The potential reason is that when $\delta = 0.0$, all return edges are ignored in call graphs, and the invoked methods are assigned with too much faultiness or successfulness scores. Especially, faultiness score plays a leading role in Op2 and can make callee methods more suspicious than the caller methods, thus decreasing the effectiveness of PRFL. Furthermore, interestingly, it can also be observed that the return edge weight does not impact PRFL effectiveness much when $\delta \geq 0.1$,

e.g., the largest improvement difference among different return edge weights is no more than 5% for all the formulas on all the subjects.

2.5.3 RQ3: Impact of Fault Number

As mentioned by existing work [124], over 82% of faults are single-faults, where the systems can be repaired by only eliminating one fault. This observation inspires us that it is plausible to apply simpler and near-optimal fault localization in most cases. In our experiment, we obtain similar observation that around 80.0% of subjects are single-fault subjects (309 out of 395 subjects in Defects4J v1.2), which also validates the prevalence of single-fault faults in software development. Hence, we classify the program faults as single-fault and multi-location-faults. Specifically, for method-level fault localization, single-fault is defined as all bug fixes located in a single method. In contrast, multi-location-faults refers to bug fixes in multiple methods. In our evaluation, we define localizing multi-location-faults as multiple tasks of single-fault localization where an ideal technique is expected to rank all faulty methods on the top.

The real faults from Defects4J include both single-fault and multi-location-faults. Table 2.9 presents the experimental results, where the left half presents the results on single-fault versions while the right half presents the results on the multi-location-faults versions. From the table, we have the following findings. First, we find that the traditional techniques perform differently on single-fault and multi-location-faults versions. For example, Dstar2 and Op2

Tech	Single-fault versions									Multi-location-faults versions								
	Top-1		Top-3		Top-5		AWE			Top-1		Top-3		Top-5		AWE		
	S	P	S	P	S	P	S	P	Impr.	S	P	S	P	S	P	S	P	Impr.
Tarantula	58	70	122	135	150	161	30.81	24.20	21.44%	17	28	59	70	79	90	27.28	20.88	23.45%
SBI	58	70	122	135	150	161	30.81	24.20	21.44%	17	27	59	70	79	90	27.28	20.89	23.41%
Ochiai	63	82	128	144	156	178	26.41	19.00	28.06%	16	28	51	68	74	96	26.22	20.43	22.08%
Jaccard	60	73	123	137	153	163	29.42	23.02	21.74%	16	26	53	67	76	91	27.27	20.66	24.26%
Ochiai2	59	71	122	136	151	162	29.83	23.64	20.76%	16	27	56	71	76	91	27.22	20.83	23.47%
Kulczynski	60	73	123	137	153	163	29.42	23.02	21.74%	16	26	53	67	76	91	27.28	20.66	24.27%
Dstar2	64	82	129	146	156	179	26.35	18.81	28.62%	15	27	48	63	65	90	27.19	20.58	24.31%
Op2	68	67	136	142	162	174	24.17	19.75	18.30%	14	16	36	43	50	57	63.98	53.23	16.81%

Column **S** and **P** indicate SBFL and PRFL respectively

Table 2.9: Overall fault Localization Results on Single-fault and Multi-location-faults of Defects4J

(marked in gray in the left half) are the two overall most effective techniques for single-fault versions, while Tarantula and SBI (marked in gray in the right half) are the two overall most effective techniques for multi-location-faults versions. In particular, Op2 performs the best on single-fault versions (also confirmed by prior work [153]), e.g., with the highest Top-1 value (i.e., 68) and the lowest AWE value (i.e., 24.17), but performs the worst on multi-location-faults versions, e.g., with the lowest Top-1 value (i.e., 14) and the highest AWE value (i.e., 63.98). The major reason might be that Op2 is specifically designed and also shown to be optimal for single-fault programs [109], but it cannot perform well for multi-location-faults programs. Second, we find that although various techniques perform differently on single-fault or multi-location-faults programs, PRFL is able to boost almost all the studied techniques similarly on both single-fault and multi-location-faults fault programs. For example, the Top-1 value improvement for Tarantula is 20.7% (from 58 to 70) on single-fault programs and 64.7% (from 17 to 28) on multi-location-faults programs. Finally, PRFL is also able to boost the originally effective techniques (e.g., Dstar2, SBI,

and Tarantula) significantly. For example, the AWE improvement for Dstar2 (i.e., 28.62%) is the highest for single-fault versions. For multi-location-faults versions, the AWE improvement for two optimal techniques, Tarantula and SBI, are 23.45% and 23.41% respectively.

2.5.4 RQ4: Comparison of PRFL and a Recently Proposed SBFL Technique

We compare PRFL with **Multric** [159], a recently proposed spectrum-based fault localization technique. Multric [159] applies pairwise learning-to-rank algorithm to ensemble multiple SBFL formulas in order to improve the accuracy of SBFL techniques. To be specific, in the experiment, we first select 34 SBFL formulas [94] to compute the suspiciousness scores for generating feature vectors for each method.

Then, we adopted LIBSVM³, a widely-used library for support vector machines, and XGBoost⁴, a widely-used optimized distributed gradient boosting library, to investigate the effectiveness. More specifically, we used RankSVM with linear kernel from LIBSVM (default settings), and LambdaRank from XGBoost (booster = `gbtree`, max_depth = 60, num_round = 100, colsample_bytree=0.85, and eta =0.5). Moreover, we performed a leave-one-out cross validation [25,94] not only across each version of the six projects, but also across the whole six projects. For the total N bugs, we can separate them into two groups: (1) one group only includes one bug is the test data for

³<https://www.csie.ntu.edu.tw/~cjlin/libsvm>

⁴<https://github.com/dmlc/xgboost>

predicting its rank; (2) and the other group with remaining $N - 1$ bugs as the training data is to build the ranking model.

Tech.	Proj.	Top1	Top3	Top5	AWE
Ochiai	Chart	6	17	19	11.83
	Lang	22	44	56	5.35
	Math	24	60	73	9.25
	Time	6	11	18	22.93
	Mockito	7	18	25	49.91
	Closure	14	29	39	108.70
	Overall	79	179	230	34.66
Multric	Chart	7	18	21	10.88
	Lang	23	47	59	6.9
	Math	21	57	69	25.17
	Time	6	14	14	39.11
	Mockito	6	14	24	48.55
	Closure	17	38	49	134.44
	Overall	80	188	236	44.18
PRFL-Ochiai	Chart	11	20	24	10.21
	Lang	30	50	58	4.76
	Math	33	65	83	6.92
	Time	7	13	18	19.93
	Mockito	9	20	32	36.66
	Closure	20	43	59	75.74
	Overall	110	212	274	25.70

Table 2.10: Effectiveness of Ochiai, Multric and PRFL

Table 2.10 presents the results of Multric, Ochiai, and PRFL. From the table, we can observe that Multric can totally locate 80/188/236 faults within Top-1/3/5, which performs slightly better than Ochiai. However, after applying PRFL, Ochiai is effectively boosted and it can rank more faults within Top-1/3/5. We can learn from the results that Multric can generate a better formula to compute suspiciousness scores. However, the program spectrum

is still the bottleneck which limits its performance. On the other hand, the improvement from PRFL suggests the benefit of using PageRank for extracting more information from coverage and call graphs.

2.5.5 RQ5: Comparison of PRFL, MBFL and the Integrated Approach

We first compare PRFL with two recent proposed mutation-based fault localization (MBFL) techniques: **Metallaxis** [116] and **MBFL-hybrid-avg** [120]. Then, we study a simple integration of PRFL and MBFL and present its performance.

Metallaxis [116] is the first mutation-based fault localization technique. Metallaxis supposes that a mutant leads to different failure outputs/messages for the failing tests, and the mutated program entity may have a high impact on the failing tests, which means it is more likely to be the root cause of the test failures. MBFL-hybrid-avg is an improved mutation-based fault localization technique which averages each statement’s MBFL suspiciousness with the suspiciousness calculated by an SBFL technique, and it performs better than other proposed techniques in [120].

In the experiment, we choose Ochiai as the basic SBFL technique for it is the most effective one on PRFL, and we customize MBFL-hybrid-avg as follows: first, to reduce the time complexity, we only mutate the methods covered by failing tests following the original work [120]; second, instead of statement-level fault localization, we average the MBFL scores of all the state-

ments within each method and use it to present the method suspiciousness. Since project **Closure** is much larger than other projects and its associated mutation testing is very time consuming, we ignore **Closure** and apply the approaches on the other five projects of Defects4J benchmark. Moreover, we choose Metallaxis as the basic MBFL technique of MBFL-hybrid-avg, and we use the following formula to compute Metallaxis score:

$$s(e) = \max_{m \in M(e)} \left(\frac{|T_f^{(m)}(e)|}{\sqrt{(|T_f^{(m)}(e)| + |T_p^{(m)}(e)|) \cdot |T_f|}} \right) \quad (2.17)$$

where $M(e)$ denotes the set of all mutants of method e , $|T_f^{(m)}(e)|$ and $|T_p^{(m)}(e)|$ denote the number of the failing and passing tests impacted by mutant m , and $|T_f|$ denotes the total number of failing tests.

Furthermore, we study a simple integration of PRFL and Metallaxis, PR-MBFL, which combines the passing and failing test weights to the native Metallaxis. Specifically, the PR-MBFL score can be computed using the following formula:

$$s(e) = \max_{m \in M(e)} \left(\frac{\sum s_f^{(m)}(e)}{\sqrt{(\sum s_f^{(m)}(e) + \sum s_p^{(m)}(e)) \cdot \sum s_f}} \right) \quad (2.18)$$

Similar to Formula 2.17, $\sum s_f^{(m)}(e)$ and $\sum s_p^{(m)}(e)$ present the sum of the weights of the failing and passing tests impacted by mutant m , $\sum s_f$ denotes the sum of the weights of all failing tests, and the test weight s_f and s_p can be extracted from Equation 2.14.

We present the experimental results of PRFL, Metallaxis, MBFL-hybrid-avg (abbreviated as MB-hy-avg) and PR-MBFL in Table 2.11. It can be ob-

served that, first, PRFL performs better than other techniques on all subjects. Second, MBFL-hybrid-avg locates less faults with in Top-1/3/5 than Metallaxis for the method-level fault localization. Third, PR-MBFL can locate more faults at Top-1, but it does not perform very on Top-3/5 compared to Metallaxis.

Tech.	Proj.	Top 1	Top 3	Top 5	AWE
Metallaxis	Chart	7	19	22	11.88
	Lang	31	55	60	3.12
	Math	19	69	84	6.77
	Time	6	11	15	18.20
	Mockito	9	19	27	52.42
	Overall	72	173	208	15.40
PR-MBFL	Chart	8	20	21	18.16
	Lang	24	48	57	4.22
	Math	20	55	68	10.53
	Time	4	9	12	18.52
	Mockito	6	13	25	52.58
	Overall	62	145	183	17.33
MB-hy-avg	Chart	7	15	22	14.00
	Lang	31	54	60	3.38
	Math	24	63	80	7.24
	Time	7	12	14	18.17
	Mockito	9	20	27	56.05
	Overall	78	164	203	16.47
PRFL	Chart	11	20	24	10.21
	Lang	30	50	58	4.76
	Math	33	65	83	6.92
	Time	7	13	18	19.93
	Mockito	9	20	32	36.66
	Overall	90	168	215	13.08

Table 2.11: Overall Results of the Comparison of PRFL, MBFL and Their Combination

Tech		STPR				PRFL			
		Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE
Overall	Tarantula	63	135	171	50.45	98	204	251	28.58
	SBI	63	135	171	50.45	97	204	251	28.58
	Ochiai	62	130	160	48.11	110	211	274	25.70
	Jaccard	62	128	160	53.00	99	203	254	27.83
	Ochiai2	61	134	164	50.72	98	206	253	28.25
	Kulczynski	62	128	160	53.00	99	203	254	27.83
	Dstar2	61	126	157	53.29	109	208	269	27.00
	Op2	62	122	160	58.38	83	184	231	62.72
Tech		HITS				SALSA			
		Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE
Overall	Tarantula	72	174	222	31.26	21	81	101	118.65
	SBI	72	174	222	31.26	21	81	101	118.65
	Ochiai	81	189	239	28.03	21	81	101	118.65
	Jaccard	74	178	226	30.65	21	81	101	118.65
	Ochiai2	74	178	225	31.04	21	81	101	118.65
	Kulczynski	74	178	226	30.65	21	81	101	118.65
	Dstar2	81	186	235	28.83	21	81	101	118.65
	Op2	72	161	198	62.33	21	81	101	118.65

Table 2.12: Overall Results of the Comparison of Link Analysis Algorithms

2.5.6 RQ6: Comparison of the Constrained PageRank and other Link Analysis Algorithms

We compare the accuracy of PRFL using different link analysis algorithms: the standard PageRank (STPR), the constrained PageRank (PRFL), HITS, and SALSA. To implement these approaches, we first construct graphs with the nodes presenting the methods/tests and edges presenting the method invocation and test coverage. Note that in this graph all nodes and edges are congeneric. Next, we generate adjacent matrices based on the graphs and implement the corresponding algorithms introduced in [49]. For HITS and SALSA, every node has authority and hub scores, and in this experiment, we only choose the authority score to present the node importance. Table 2.12 shows the results of the approaches with different link analysis algorithms,

and we can observe that the constrained PageRank outperforms the standard PageRank, which validates the necessity of edge differentiation. On the other hand, HITS and SALSA perform poorly compared to the constrained PageRank. Such results inspire us that it is necessary for the vanilla link analysis algorithms to be customized for fault localization. Interestingly, SALSA computes the same score on all the formulas. We study the intermediate results and find that all the methods share the same faultiness and successfulness scores, because of the graphs and adjacent matrices of failing and passing tests are symmetric, and SALSA sets the same score to all nodes.

2.5.7 RQ7: Effectiveness of PRFL+

We study the effectiveness of PRFL+ by analyzing the impact of the dynamic call graph, passing test weight adjustment, and test and method reduction, respectively.

2.5.7.1 Impact of Dynamic Call Graph

We design experiments in order to verify whether adopting dynamic call graphs would impact on Top-N rank, that is presented in Table 2.13. The results show that using dynamic call graphs instead of static call graphs makes no obvious changes in Top-N. For instance, in row Ochiai, PRFL with static call graphs captures 110 faults while PRFL with dynamic call graphs captures 109 faults. On the other hand, PRFL can perform slightly better on Tarantula using dynamic call graphs, which can localize 2 more faults on Top-3 (205

vs. 206) and Top-5 (251 vs. 253). We suspect the reason to be two-fold: (1) the majority of static call graph edges and nodes are also in the dynamic call graphs, (2) call graphs are used to fine-tune the final fault localization results and may not impact the results significantly.

Tech	Top-1		Top-3		Top-5	
	S	D	S	D	S	D
Tarantula	98	98	205	206	251	253
SBI	97	97	205	206	251	251
Ochiai	110	109	212	212	265	265
Jaccard	99	99	204	204	254	254
Ochiai2	98	98	207	207	253	253
Kulczynski	99	99	204	204	254	254
Dstar2	109	109	209	209	269	269
Op2	83	83	185	185	231	231

Column **S** and **D** indicate the static
and dynamic call graph

Table 2.13: Overall Results of Dynamic Call Graph

2.5.7.2 Impact of Passing Test Weight Adjustment

We extend our experiments to explore the performance of passing test weight adjustment. Table 2.14 shows the fault localization results of PRFL without passing test weight adjustment. Specifically, the columns represent the evaluated metrics (i.e., Top-1/3/5, and AWE). For each metric, Column **P** represents PRFL without passing test weight adjustment and Column **WA** represents PRFL with passing test weight adjustment. The rows represent the subjects with the eight fault localization formulas. The overall results of the six Defects4J subjects are displayed in the bottom row of the table.

Tech		Top-1		Top-3		Top-5		AWE		
		P	WA	P	WA	P	WA	P	WA	impr.
Chart	Tarantula	12	12	21	21	24	24	13.63	13.67	-0.28%
	SBI	11	11	21	21	24	24	13.63	13.67	-0.28%
	Ochiai	11	11	20	20	24	24	10.21	10.21	0.00%
	Jaccard	11	11	20	20	24	24	10.60	10.60	0.00%
	Ochiai2	12	12	21	21	24	24	11.40	11.37	0.34%
	Kulczynski	11	11	20	20	24	24	10.60	10.60	0.00%
	Dstar2	10	10	21	21	24	24	9.83	9.83	0.00%
	Op2	7	7	17	17	20	20	61.37	61.37	0.00%
Lang	Tarantula	27	27	50	50	57	57	5.26	5.25	0.29%
	SBI	27	27	50	50	57	57	5.26	5.25	0.29%
	Ochiai	30	30	51	51	58	58	4.76	4.76	0.00%
	Jaccard	28	28	50	50	57	57	4.88	4.87	0.31%
	Ochiai2	28	28	50	50	57	57	5.33	5.32	0.29%
	Kulczynski	28	28	50	50	57	57	4.88	4.87	0.31%
	Dstar2	30	30	50	50	58	58	4.65	4.64	0.33%
	Op2	29	29	48	48	56	56	4.61	4.59	0.33%
Math	Tarantula	32	32	65	65	78	79	7.36	7.28	1.03%
	SBI	32	32	65	65	78	79	7.36	7.28	1.03%
	Ochiai	33	33	65	66	83	83	6.92	6.85	0.95%
	Jaccard	32	32	66	66	79	80	7.33	7.25	1.03%
	Ochiai2	31	31	67	67	79	80	7.35	7.29	0.90%
	Kulczynski	32	32	66	66	79	80	7.33	7.25	1.03%
	Dstar2	33	34	65	65	83	83	6.91	6.84	0.96%
	Op2	28	28	61	61	76	76	8.35	8.32	0.45%
Time	Tarantula	7	7	14	14	16	16	23.30	23.26	0.16%
	SBI	7	7	14	14	16	16	23.30	23.26	0.16%
	Ochiai	7	7	13	13	18	18	19.93	19.96	-0.19%
	Jaccard	7	7	12	12	17	17	23.44	23.33	0.47%
	Ochiai2	7	7	14	14	17	16	23.33	23.22	0.48%
	Kulczynski	7	7	12	12	17	17	23.44	23.33	0.47%
	Dstar2	7	7	12	12	12	12	21.48	21.59	-0.52%
	Op2	4	4	11	11	13	13	60.19	60.22	-0.06%
Closure	Tarantula	13	14	33	34	45	46	87.01	86.81	0.22%
	SBI	13	14	33	34	45	46	87.02	86.83	0.22%
	Ochiai	20	23	43	44	59	59	75.74	75.62	0.16%
	Jaccard	14	15	34	36	47	47	86.29	86.03	0.30%
	Ochiai2	13	14	33	36	46	47	86.83	86.59	0.29%
	Kulczynski	14	15	34	36	47	47	86.29	86.03	0.30%
	Dstar2	21	23	42	44	59	59	74.77	74.63	0.19%
	Op2	10	10	32	32	43	42	87.76	87.74	0.03%
Mockito	Tarantula	7	7	22	21	31	31	34.89	34.74	0.45%
	SBI	7	7	22	21	31	31	34.89	34.74	0.45%
	Ochiai	9	9	20	20	32	32	36.66	36.34	0.86%
	Jaccard	7	7	22	22	30	31	34.42	34.08	0.99%
	Ochiai2	7	7	22	22	30	31	35.24	34.89	0.97%
	Kulczynski	7	7	22	22	30	31	34.42	34.08	0.99%
	Dstar2	8	8	19	19	33	33	44.34	44.05	0.65%
	Op2	5	5	16	16	23	23	154.03	153.87	0.10%
Overall	Tarantula	98	99	205	205	251	253	28.58	28.50	0.04%
	SBI	97	98	205	205	251	253	28.58	28.50	0.04%
	Ochiai	110	113	212	214	274	274	25.70	25.62	0.09%
	Jaccard	99	100	204	206	254	256	27.83	27.69	0.05%
	Ochiai2	98	99	207	210	253	255	28.25	28.11	0.08%
	Kulczynski	99	100	204	206	254	256	27.83	27.69	0.05%
	Dstar2	109	112	209	211	269	269	27.00	26.93	0.06%
	Op2	83	83	185	185	231	230	62.72	62.68	0.00%

Weight Modification effectiveness

Legend deterioration improvement

Column **P** and **WA** indicate PRFL and PRFL with Passing Test Weight Adjustment respectively

Table 2.14: Results of Passing Test Weight Adjustment

From this table, it can be generally observed that passing test weight adjustment does not incur significant differences for PRFL analysis. For instance, for Ochiai, that is the most effective SBFL technique for PRFL analysis, PRFL can locate 110 out of 395 faults within Top-1, while PRFL with passing weight adjustment can locate 113, which slightly improves the performance of PRFL.

Method Name	Failing Score		Passing Score	
	PRFL	WA	PRFL	WA
*.Fraction:getNumerator()I	0.484206	0.484206	0.015727	0.015912
*.BigFraction:getDenominatorAsInt()I	0.486999	0.486999	0.007583	0.007799
*.BigFraction:<init>(DDII)V	0.511939	0.511939	0.003186	0.003262
*.util.FastMath:floor(D)D	0.978647	0.978647	0.366248	0.366175
*.util.FastMath:abs(J)J	0.487929	0.487929	0.440933	0.439075

Table 2.15: Result Snippet of Math-1

We intercept a resulting snippet from Math-1 in Table 2.15, where the SBFL scores of PRFL without passing test weight adjustment are displayed. From Table 2.15, it can be observed that the failing score for each method is not affected since the weights of failing tests are not tuned. On the other hand, after modifying the weights on passing tests, the most scores are changed by lower than 10^{-3} . The SBFL scores can be subsequently computed in PRFL based on weighted spectra. Therefore, we can conclude that the passing test weight adjustment has only slight impact on PRFL.

2.5.7.3 Impact of Test and Method Reduction

The efficacy of reducing tests and methods for PRFL analysis are demonstrated in Table 2.16 and 2.17, respectively. In both tables, the columns

present the evaluated metrics in Top-N and AWE of PRFL without test and method reduction, and the rows represent the subjects and fault localization formulas. In the bottom rows of both tables, we present the overall results for six Defects4J subjects. Moreover, the performance improvement and deteriorations by adopting test and method reduction is marked being dark and light gray, respectively.

Impact of Test-based Reduction. From Table 2.16, it can be observed that the overall performance of fault localization is changed slightly. For instance, for PRFL with test-based reduction, Ochiai2 and Op2 deteriorate, and Jaccard, Kulczynski, and Dstar improve at Top-1, within a slight range, compared with the original PRFL. The changes on AWE are mostly limited within 2%.

Impact of Method-based Reduction. From Table 2.17, it can be shown that similar to the test-based reduction approach, the method-based reduction approach does not incur significant changes on Top-N and AWE.

The Number of Tests and Methods after Pruning. We further explore the average number of remaining tests and methods after pruning, as shown in Table 2.18. We design a metric, namely *reducing rate*, to present the level of the reducing the number of tests and methods after applying pruning. Specifically, the *reducing rate* $R = (1 - \frac{F}{I}) \times 100\%$, where I and F are the numbers of the original and remained tests/methods after pruning, respectively.

In Table 2.18, the left part of the table presents the results of the number of the remaining tests after applying the test-based reduction approach

Tech		Top-1		Top-3		Top-5		AWE		
		P	TR	P	TR	P	TR	P	TS	impr.
Chart	Tarantula	12	12	21	20	24	24	13.63	13.67	-0.28%
	SBI	11	11	21	20	24	24	13.63	13.67	-0.28%
	Ochiai	11	11	20	20	24	24	10.21	10.37	-1.51%
	Jaccard	11	11	20	20	24	23	10.60	10.90	-2.90%
	Ochiai2	12	11	21	20	24	25	11.40	11.13	2.36%
	Kulczynski	11	11	20	20	24	23	10.60	10.90	-2.90%
	Dstar2	10	9	21	20	24	22	9.83	11.17	-13.70%
	Op2	7	6	17	17	20	20	61.37	61.40	-0.06%
Lang	Tarantula	27	27	50	50	57	57	5.26	5.28	-0.29%
	SBI	27	27	50	50	57	57	5.26	5.28	-0.29%
	Ochiai	30	29	51	50	58	58	4.76	4.72	0.97%
	Jaccard	28	29	50	50	57	58	4.88	4.78	2.20%
	Ochiai2	28	26	50	50	57	57	5.33	5.08	4.76%
	Kulczynski	28	29	50	50	57	58	4.88	4.78	2.20%
	Dstar2	30	30	50	49	58	56	4.65	4.82	-3.64%
	Op2	29	29	48	48	56	56	4.61	4.73	-2.67%
Math	Tarantula	32	32	65	65	78	79	7.36	7.32	0.51%
	SBI	32	32	65	65	78	79	7.36	7.32	0.51%
	Ochiai	33	34	65	66	83	84	6.92	6.88	0.55%
	Jaccard	32	33	66	67	79	80	7.33	7.27	0.77%
	Ochiai2	31	32	67	68	79	79	7.35	7.30	0.77%
	Kulczynski	32	33	66	67	79	80	7.33	7.27	0.77%
	Dstar2	33	34	65	64	83	84	6.91	6.86	0.68%
	Op2	28	28	61	60	76	76	8.35	8.33	0.23%
Time	Tarantula	7	7	14	14	16	16	23.30	23.30	0.00%
	SBI	7	7	14	14	16	16	23.30	23.30	0.00%
	Ochiai	7	7	13	12	18	17	19.93	20.11	-0.93%
	Jaccard	7	7	12	12	17	17	23.44	23.41	0.16%
	Ochiai2	7	7	14	14	17	17	23.33	23.26	0.32%
	Kulczynski	7	7	12	12	17	17	23.44	23.41	0.16%
	Dstar2	7	7	12	12	12	12	21.48	21.70	-1.03%
	Op2	4	4	11	11	13	13	60.19	60.19	0.00%
Closure	Tarantula	13	13	33	33	45	45	87.01	87.00	0.01%
	SBI	13	13	33	33	45	45	87.02	87.02	0.01%
	Ochiai	20	20	43	43	59	59	75.74	75.73	0.01%
	Jaccard	14	14	34	34	47	47	86.29	86.27	0.03%
	Ochiai2	13	13	33	33	46	46	86.83	86.80	0.04%
	Kulczynski	14	14	34	34	47	47	86.29	86.27	0.03%
	Dstar2	21	22	42	42	59	59	74.77	74.74	0.04%
	Op2	10	10	32	32	43	43	87.76	87.75	0.01%
Mockito	Tarantula	7	7	22	22	31	31	34.89	34.84	0.15%
	SBI	7	7	22	22	31	31	34.89	34.84	0.15%
	Ochiai	9	9	20	20	32	32	36.66	36.58	0.22%
	Jaccard	7	7	22	22	30	30	34.42	34.39	0.08%
	Ochiai2	7	7	22	22	30	30	35.24	35.24	0.00%
	Kulczynski	7	7	22	22	30	30	34.42	34.39	0.08%
	Dstar2	8	8	19	19	33	33	44.34	44.39	-0.12%
	Op2	5	5	16	16	23	23	154.03	154.08	-0.03%
Overall	Tarantula	98	98	205	204	251	252	28.58	28.57	0.03%
	SBI	97	97	205	204	251	252	28.58	28.57	0.03%
	Ochiai	110	110	212	211	274	274	25.70	25.73	-0.11%
	Jaccard	99	101	204	205	254	255	27.83	27.84	-0.03%
	Ochiai2	98	96	207	207	253	254	28.25	28.13	0.41%
	Kulczynski	99	101	204	205	254	255	27.83	27.84	-0.03%
	Dstar2	109	110	209	206	269	266	27.00	27.28	-1.06%
	Op2	83	82	185	184	231	231	62.72	62.75	-0.05%

Effectiveness of Test-based Reduction

Legend deterioration improvement

Column **P** and **TR** indicate PRFL and PRFL with Test-based Reduction respectively

Table 2.16: Results of Test-based Reduction

Tech		Top-1		Top-3		Top-5		AWE		
		P	MR	P	MR	P	MR	P	MR	impr.
Chart	Tarantula	12	11	21	21	24	24	13.63	13.75	-0.85%
	SBI	11	10	21	21	24	24	13.63	13.75	-0.85%
	Ochiai	11	10	20	21	24	24	10.21	10.17	0.38%
	Jaccard	11	10	20	20	24	23	10.60	10.98	-3.63%
	Ochiai2	12	10	21	20	24	24	11.40	11.37	0.34%
	Kulczynski	11	10	20	20	24	23	10.60	10.98	-3.63%
	Dstar2	10	8	21	20	24	22	9.83	11.17	-13.70%
	Op2	7	7	17	17	20	20	61.37	61.33	0.06%
Lang	Tarantula	27	26	50	50	57	57	5.26	5.22	0.73%
	SBI	27	26	50	50	57	57	5.26	5.22	0.73%
	Ochiai	30	28	51	49	58	57	4.76	4.75	0.16%
	Jaccard	28	28	50	49	57	57	4.88	4.83	1.10%
	Ochiai2	28	25	50	49	57	57	5.33	5.07	4.91%
	Kulczynski	28	28	50	49	57	57	4.88	4.83	1.10%
	Dstar2	30	29	50	49	58	55	4.65	4.86	-4.46%
	Op2	29	28	48	48	56	56	4.61	4.69	-1.84%
Math	Tarantula	32	34	65	68	78	79	7.36	7.16	2.76%
	SBI	32	34	65	68	78	79	7.36	7.16	2.76%
	Ochiai	33	34	65	65	83	82	6.92	6.87	0.75%
	Jaccard	32	34	66	69	79	80	7.33	7.11	2.90%
	Ochiai2	31	33	67	70	79	79	7.35	7.16	2.69%
	Kulczynski	32	34	66	69	79	80	7.33	7.11	2.90%
	Dstar2	33	34	65	65	83	81	6.91	6.96	-0.75%
	Op2	28	29	61	61	76	73	8.35	8.55	-2.32%
Time	Tarantula	7	6	14	14	16	16	23.30	23.33	-0.16%
	SBI	7	6	14	14	16	16	23.30	23.33	-0.16%
	Ochiai	7	6	13	12	18	17	19.93	20.04	-0.56%
	Jaccard	7	6	12	12	17	17	23.44	23.37	0.32%
	Ochiai2	7	6	14	14	17	17	23.33	23.19	0.63%
	Kulczynski	7	6	12	12	17	17	23.44	23.37	0.32%
	Dstar2	7	6	12	11	12	12	21.48	21.89	-1.90%
	Op2	4	4	11	11	13	13	60.19	60.19	0.00%
Closure	Tarantula	13	14	33	35	45	46	87.01	86.52	0.56%
	SBI	13	14	33	35	45	46	87.02	86.54	0.56%
	Ochiai	20	23	43	44	59	59	75.74	75.65	0.11%
	Jaccard	14	15	34	37	47	47	86.29	85.69	0.70%
	Ochiai2	13	14	33	37	46	46	86.83	86.28	0.64%
	Kulczynski	14	15	34	37	47	47	86.29	85.69	0.70%
	Dstar2	21	25	42	43	59	58	74.77	74.65	0.17%
	Op2	10	10	32	31	43	42	87.76	87.58	0.20%
Modkito	Tarantula	7	7	22	22	31	31	34.89	34.88	0.04%
	SBI	7	7	22	22	31	31	34.89	34.88	0.04%
	Ochiai	9	9	20	20	32	31	36.66	36.67	-0.04%
	Jaccard	7	7	22	22	30	30	34.42	34.36	0.19%
	Ochiai2	7	7	22	22	30	30	35.24	35.22	0.04%
	Kulczynski	7	7	22	22	30	30	34.42	34.36	0.19%
	Dstar2	8	8	19	19	33	32	44.34	44.43	-0.21%
	Op2	5	5	16	16	23	23	154.03	154.01	0.01%
Overall	Tarantula	98	98	205	210	251	253	28.58	28.48	0.34%
	SBI	97	97	205	210	251	253	28.58	28.48	0.34%
	Ochiai	110	110	212	211	274	270	25.70	25.69	0.04%
	Jaccard	99	100	204	209	254	254	27.83	27.72	0.38%
	Ochiai2	98	95	207	212	253	253	28.25	28.05	0.72%
	Kulczynski	99	100	204	209	254	254	27.83	27.72	0.38%
	Dstar2	109	110	209	207	269	260	27.00	27.33	-1.22%
	Op2	83	83	185	184	231	227	62.72	62.72	-0.01%

Legend  deterioration  improvement

Column P and MR indicate PRFL and PRFL with Method-based Reduction respectively

Table 2.17: Results of Method-based Reduction

Test-based Reduction				Method-based Reduction			
Project	Initial	Final	Reducing Rate	Project	Initial	Final	Reducing Rate
Chart	1821	628	65.50%	Chart	4473	103	97.70%
Lang	1856	55	97.00%	Lang	1994	10	99.50%
Math	2871	1064	63.00%	Math	3871	43	98.89%
Time	3924	3432	12.50%	Time	3442	122	96.46%
Closure	7187	6993	2.71%	Closure	7260	780	89.26%
Mockito	1136	1113	2.03%	Mockito	1171	169	85.57%

Table 2.18: Average Test (Left) and Method (Right) Number after Reduction

in terms of the six projects. It indicates that the test-based reduction leads to a different number of the remaining tests among different projects. For instance, for project Lang, the *reducing rate* is around 97%, while the *reducing rate* in project Closure is only 2.71%. Moreover, the right part of the table presents the results of the remaining number of the methods after applying the method-based reduction approach. It can be observed that the *reducing rates* of methods of all the projects are higher than their corresponding *reducing rates* of tests, ranging from 85% to 99%.

Time Efficiency. Reducing the size of the transition matrices is expected to reduce the computational cost of PRFL. Therefore, we collect the computation time for PRFL without applying the test and method reduction approaches to verify how exactly the computational cost can be optimized, as in Table 2.19. It can be shown that both the test and method reduction approaches result in shorter computation time than PRFL.

Table 2.19 shows the average computation time in each phase of PRFL, the test-based reduction approach, and the method-based reduction approach. Column **DP**, Column **Analysis** and Column **Ranking** represent data processing

Tech	Sub	DP	Analysis	Ranking	Total	Time Radio
PRFL	Chart	6.49	1.33	0.01	109.72	100.00%
	Closure	888.87	6.61	0.01	1558.95	100.00%
	Lang	0.81	0.38	0.01	47.31	100.00%
	Math	8.33	1.47	0.01	384.34	100.00%
	Mockito	4.30	0.15	0.01	88.76	100.00%
	Time	19.39	1.32	0.01	68.00	100.00%
Test-based	Chart	2.52	0.45	0.01	104.87	95.58%
	Closure	835.75	6.56	0.01	1505.76	96.59%
	Lang	0.01	-	0.01	46.133	97.51%
	Math	3.22	0.50	0.01	378.26	98.42%
	Mockito	4.25	0.14	0.01	88.70	99.93%
	Time	17.4	1.14	0.01	65.83	96.80%
Method-based	Chart	0.01	0.01	0.01	102.09	93.04%
	Closure	279.12	0.46	0.01	943.03	60.49%
	Lang	0.01	-	0.01	46.13	97.51%
	Math	0.18	0.01	0.01	374.73	97.49%
	Mockito	1.49	0.02	0.01	85.82	96.68%
	Time	2.78	0.03	0.01	50.10	73.68%

Table 2.19: Computation Time of PRFL, Test-based Reduction and Method-based Reduction

of PRFL(i.e., transition matrices construction), PageRank analysis, and suspiciousness computation/ranking, respectively. Since the time of test coverage and call graph collection is not affected by the test and method reduction, we do not present it in Table 2.19 and the readers can find it in Table 2.6(Column COV and CG). The main differences are presented in Column DP and Column Analysis. "-" represents the time taken by this phase is negligible (far below 0.01s). It can be shown that both test- and method-based reduction approaches are timely efficient compared with PRFL. Specifically, Chart, Lang, and Math using the test-based reduction approach reduce less than 2.5% computation time of the original PRFL. The method-based reduction approach can reduce more computation time, resulting in around 40% and 27% computation time of the original PRFL for Closure and Time.

2.5.8 RQ8: Effectiveness of PRFL_{MA}

In order to study the effectiveness of PRFL_{MA}, we add an additional experiment to study the performance of MA combined with vanilla SBFL, namely SBFL_{MA}. Table 2.20 shows the impact of SBFL, PRFL, SBFL_{MA} and PRFL_{MA} for 8 SBFL formulas.

From Table 2.20, the overall results suggest that SBFL_{MA} performs much better than the vanilla SBFL, further conforming the results of prior work [142]. For instance, on Closure, SBFL_{MA} can locate 33 faults within Top-1 while the vanilla SBFL can only locate 14 faults within Top-1 by using Dstar2.

We also observe that PRFL_{MA} can locate even more faults compared with PRFL and SBFL_{MA}. PRFL_{MA} successfully localizes 137 and 133 at Top-1, with Ochiai and Dstar2 as the overall two most effective SBFL techniques. Compared to the state-of-the-art SBFL techniques, PRFL_{MA} can rank 73.4%, 49.2%, and 31.3% more faults at Top-1, Top-3 and Top-5, respectively.

2.5.9 RQ9: Impact of Fault Type

So far we have studied the effectiveness of PRFL, SBFL_{MA} and PRFL_{MA} on real faults from the Defects4J dataset. In this section, we further study the effectiveness of the above techniques on artificial mutation faults from other open-source projects to reduce the threats to external validity.

Table 2.21 presents our results on 96925 mutation faults from 240 GitHub projects. In the table, different rows present the results of different

fault localization formulas, while different columns present the different metrics used.

According to the table, we find that the most effective techniques of SBFL are the same with those for single real faults on Defects4J (shown in Table 2.9), i.e., Dstar2 and Op2 (marked in gray in the table), demonstrating the consistent result on single real and mutation faults (Note that mutation faults are all single faults since PIT only makes one syntactic modification for each mutant).

The most effective techniques of PRFL on artificial mutation faults are Ochiai and Dstar2, which are the same with those on Defects4J. PRFL can locate 27378, 49253 and 59547 faults within Top-1/3/5. Compared to the state-of-the-art SBFL techniques, PRFL can rank 82.3% (24613 vs. 13505) more faults at Top-1, and reduce the AWE values from 7.59 to 6.82 (10.1% more precise) on Op2.

PRFL performs better than SBFL_{MA} on artificial mutation faults at Top-1, 3 and 5. However, it is worth noting that the AWE values of SBFL_{MA} are lower than PRFL. According to Section 2.3.3, we infer that MA enables the faulty method to have a higher SBFL score than other methods in most cases.

PRFL_{MA} can locate more faults compared with PRFL and SBFL_{MA} on artificial mutation faults, with Ochiai and Ochiai2 as the two most effective SBFL techniques. PRFL_{MA} successfully locates 35058 faults within Top-1, that

is 36% of total mutation faults. Compared with PRFL and SBFL_{MA}, PRFL_{MA} can further rank 28% (35058 vs. 27231) and 87% (35058 vs. 18703) more faults at Top-1.

2.5.10 RQ10: Statistical Significance

To further validate the effectiveness of the proposed approaches, we use paired t -test [68, 129] to investigate the statistical significance of the experimental results. Specifically, we regard each subject in Defects4J dataset as an individual item, and group its experimental results (e.g. Top 1 scores) of two different techniques (e.g. SBFL and PRFL) as a pair. Next, the paired t -test is applied on such pairs to compute the p -values to analyze the significance of the difference between the two approaches.

We present the results of paired t -test in Table 2.22. From the table, it can be observed that the p -values of most ranking formulas are less than 0.05 of Top 1, 3 and 5 (except Op2), and these results suggest the statistical significance that PRFL and Method-level Aggregation can boost most SBFL techniques in term of the accuracy of fault localization on Top N.

Tech	SBFL				PRFL				SBFL _{MA}				PRFL _{MA}				
	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE	
Chart	Tarantula	7	20	22	14.92	12	21	24	13.63	7	23	25	10.10	12	24	27	9.98
	SBI	7	20	22	14.92	11	21	24	13.63	7	23	25	10.10	11	24	27	9.98
	Ochiai	6	17	19	11.83	11	20	24	10.21	7	19	21	9.37	12	23	26	7.52
	Jaccard	6	17	20	12.12	11	20	24	10.60	7	19	22	9.02	12	22	25	7.37
	Ochiai2	6	17	21	12.15	12	21	24	11.40	7	19	23	8.37	12	24	27	7.02
	Kulczynski	6	17	20	12.12	11	20	24	10.60	7	19	22	9.02	12	22	25	7.37
	Dstar2	5	16	19	13.65	10	21	24	9.83	6	18	21	11.25	11	23	27	7.52
	Op2	5	14	16	66.19	7	17	20	61.37	6	16	19	62.90	7	16	20	60.67
Lang	Tarantula	21	45	57	5.81	27	49	57	5.26	26	49	59	4.28	29	53	61	4.11
	SBI	21	45	57	5.81	27	49	57	5.26	26	49	59	4.28	29	53	61	4.12
	Ochiai	22	44	56	5.35	30	50	58	4.76	28	48	58	3.85	34	54	61	3.55
	Jaccard	22	44	56	5.39	28	49	57	4.88	28	47	58	3.95	32	53	61	3.72
	Ochiai2	21	45	55	5.35	28	49	57	5.33	27	48	57	3.86	32	53	61	3.96
	Kulczynski	22	44	56	5.39	28	49	57	4.88	28	47	58	3.95	32	53	61	3.72
	Dstar2	23	45	55	5.33	30	49	58	4.65	29	49	55	4.02	34	53	61	3.44
	Op2	23	45	56	5.48	29	47	56	4.61	29	46	56	4.15	36	48	56	3.76
Math	Tarantula	24	61	73	8.89	32	65	78	7.36	30	79	92	5.25	38	80	90	5.09
	SBI	24	61	73	8.87	32	65	78	7.36	30	79	92	5.25	37	80	90	5.09
	Ochiai	24	60	73	9.25	33	65	83	6.92	31	74	92	5.17	40	83	90	4.63
	Jaccard	24	61	73	8.87	32	66	79	7.33	32	75	92	5.28	38	79	91	5.11
	Ochiai2	24	61	73	8.88	31	67	79	7.35	30	77	92	5.27	38	81	91	5.09
	Kulczynski	24	61	73	8.87	32	66	79	7.33	32	75	92	5.28	38	79	91	5.11
	Dstar2	24	60	73	9.40	33	65	83	6.91	31	74	91	5.24	40	82	90	4.64
	Op2	23	54	65	10.66	28	61	76	8.35	29	67	80	7.79	33	62	77	7.99
Time	Tarantula	5	11	16	25.37	7	14	16	23.30	6	14	21	15.57	7	17	20	14.30
	SBI	5	11	16	25.37	7	14	16	23.30	6	14	21	15.57	7	17	20	14.30
	Ochiai	6	11	18	22.93	7	13	18	19.93	7	14	18	15.06	8	17	21	13.33
	Jaccard	5	9	18	25.41	7	12	17	23.44	7	13	20	15.83	8	16	21	14.33
	Ochiai2	5	11	16	25.44	7	14	17	23.33	7	15	22	15.43	8	17	21	14.44
	Kulczynski	5	9	18	25.41	7	12	17	23.44	7	13	20	15.83	8	16	21	14.33
	Dstar2	6	11	12	25.22	7	12	12	21.48	7	14	16	18.39	7	14	16	16.00
	Op2	8	12	14	62.67	4	11	13	60.19	7	13	15	63.13	4	11	11	63.56
Closure	Tarantula	12	26	36	120.30	13	33	45	87.01	27	57	66	69.44	28	58	65	69.32
	SBI	12	26	36	120.30	13	33	45	87.02	27	57	66	69.44	28	58	65	69.32
	Ochiai	14	29	39	108.70	20	43	59	75.74	31	64	70	62.17	32	63	69	62.04
	Jaccard	13	27	37	119.40	14	34	47	86.29	30	62	68	68.67	31	61	67	68.54
	Ochiai2	13	26	37	120.01	13	33	46	86.83	30	61	68	69.10	30	60	67	68.94
	Kulczynski	13	27	37	119.40	14	34	47	86.29	30	62	68	68.67	31	61	67	68.54
	Dstar2	14	28	39	108.23	21	42	59	74.77	33	63	70	61.72	32	62	69	61.65
	Op2	17	33	42	119.26	10	32	43	87.76	32	59	61	124.32	32	59	61	124.19
Mockito	Tarantula	6	18	25	39.51	7	22	31	34.89	10	26	29	25.07	12	26	31	22.01
	SBI	6	18	25	39.51	7	22	31	34.89	10	26	29	25.07	12	26	31	22.01
	Ochiai	7	18	25	49.91	9	20	32	36.66	11	26	30	38.12	11	27	35	28.25
	Jaccard	6	18	25	39.32	7	22	30	34.42	10	26	30	25.92	12	26	31	22.36
	Ochiai2	6	18	25	39.46	7	22	30	35.24	10	26	29	25.33	12	26	31	22.51
	Kulczynski	6	18	25	39.32	7	22	30	34.42	10	26	30	25.92	12	26	31	22.36
	Dstar2	7	17	23	57.96	8	19	33	44.34	10	25	28	47.38	9	26	36	37.25
	Op2	6	14	19	155.43	5	16	23	154.03	10	19	23	151.86	7	19	25	151.96
Overall	Tarantula	75	181	229	35.80	98	204	251	28.58	106	248	292	21.62	126	258	294	20.80
	SBI	75	181	229	35.80	97	204	251	28.58	106	248	292	21.62	124	258	294	20.80
	Ochiai	79	179	230	34.66	110	211	274	25.70	115	245	289	22.29	137	267	302	19.89
	Jaccard	76	176	229	35.08	99	203	254	27.83	114	242	290	21.45	133	257	296	20.24
	Ochiai2	75	178	227	35.21	98	206	253	28.25	111	246	291	21.22	132	261	298	20.33
	Kulczynski	76	176	229	35.08	99	203	254	27.83	114	242	290	21.45	133	257	296	20.24
	Dstar2	79	177	221	36.63	109	208	269	27.00	116	243	281	24.67	133	260	299	21.75
	Op2	82	172	212	69.95	83	184	231	62.72	113	220	254	69.02	119	215	250	68.69

Table 2.20: Result of SBFL,PRFL,SBFL_{MA} and PRFL_{MA} on all Defects4J bugs

Tech	SBFL				PRFL				SBFL _{MA}				PRFL _{MA}			
	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE
Tarantula	12026	34891	45460	10.61	23327	45491	55044	9.35	18688	44365	55546	6.00	32561	55053	64482	4.95
SBI	10871	32704	43549	10.76	19728	42150	52263	9.64	16613	41885	52625	6.15	27269	51057	61594	5.23
Ochiai	13113	37500	49096	8.05	27231	48953	59146	6.76	18671	44387	55568	6.00	34960	55460	63181	5.03
Jaccard	13072	37334	48919	8.35	27088	48619	58863	7.07	18671	44387	55568	6.00	34928	55442	63121	5.05
Ochiai2	12976	36804	48166	9.40	26389	47381	57215	8.51	18703	44387	55568	6.00	35058	55544	63263	4.98
Kulczynski	13072	37336	48919	8.35	27088	48619	58863	7.07	18671	44387	55568	6.00	34928	55442	63121	5.05
Dstar2	13179	37667	49329	7.92	27378	49253	59547	6.57	18671	44387	55568	6.00	34642	55249	62873	5.10
Op2	13505	38288	49915	7.59	24613	45940	56298	6.82	18228	43784	54929	6.02	23970	42743	50323	6.08

Table 2.21: Fault localization Results on Mutation Faults

Comp.	Tech.	Top 1	Top 3	Top 5	AWE
SBFL vs PRFL	Tarantula	0.0119*	0.0024♦	0.0278*	0.1134
	SBI	0.0131*	0.0024♦	0.0278*	0.1135
	Ochiai	0.0053♦	0.0174*	0.0268*	0.0715
	Jaccard	0.0119*	0.0004◇	0.0230*	0.1111
	Ochiai2	0.0160*	0.0003◇	0.0077♦	0.1219
	Kulczynski	0.0119*	0.0004◇	0.0230*	0.1110
	Dstar2	0.0073♦	0.0204*	0.0197*	0.0590
	Op2	0.4697	0.0798	0.0675	0.0994
PRFL vs PRFL _{MA}	Tarantula	0.0494*	0.0300*	0.0324*	0.0171*
	SBI	0.0534	0.0300*	0.0324*	0.0171*
	Ochiai	0.0257*	0.0151*	0.0075♦	0.0151*
	Jaccard	0.0330*	0.0354*	0.0360*	0.0182*
	Ochiai2	0.0363*	0.0343*	0.0303*	0.0153*
	Kulczynski	0.0330*	0.0354*	0.0360*	0.0182*
	Dstar2	0.0356*	0.0216*	0.0041♦	0.0172*
	Op2	0.0688	0.1466	0.1709	0.1876
SBFL _{MA} vs PRFL _{MA}	Tarantula	0.0153*	0.0211*	0.3397	0.0767
	SBI	0.0117*	0.0211*	0.3397	0.0771
	Ochiai	0.0265*	0.0265*	0.0683	0.0878
	Jaccard	0.0075♦	0.0321*	0.1146	0.0393*
	Ochiai2	0.0181*	0.0321*	0.1509	0.0500*
	Kulczynski	0.0075♦	0.0321*	0.1146	0.0393*
	Dstar2	0.0764	0.0495*	0.0665	0.0591
	Op2	0.2812	0.2171	0.2580	0.2167

* $p < 0.05$, ♦ $p < 0.01$, ◇ $p < 0.001$

Table 2.22: Paired t -test Results of Different Techniques

2.6 Threats to Validity

The main threats to the validity of our work consist of three parts: internal, external, and construct validity threat.

Threats to internal validity are mainly concerned with uncontrolled factors. These factors may impact the results and reduce their creditability. In this work, the main threat to internal validity is the potential defects in the implementation of our own technique and the reimplementing of other baselines. To reduce this threat, we utilize state-of-the-art tools and frameworks, such as Java Agent, ASM bytecode manipulation framework and Numpy library to build our technique. Moreover, we also reimplement other baseline techniques by following their papers. We carefully review all our code and experiment scripts to ensure their correctness. However, there is always a small possibility of defects, which introduces risk to the result’s correctness.

Threats to external validity are mainly concerned with whether the performance of our techniques can still hold in other experimental settings. In our work, the subjects (including tests and faults) may introduce threats to external validity. To reduce these threats, we use the real-world subjects from Defects4J and Bugs.jar dataset to evaluate our approach, respectively. However, the total subject number is relatively small and it may limit the generalizability of our technique. We further utilize a bunch of artificial faults to evaluate the effectiveness of our approach. However, the artificial faults are generated by mutation testing tool, and this will make the diversity of artificial faults limited. Therefore, although the experimental results of artificial faults

have low variance (due to a large number of mutants), they may be biased. To further reduce the threats, we are planning to collect more real-world subjects to evaluate our technique.

One of the threats to construct validity is the suitability of our evaluation metrics. To reduce this risk, we follow the suggestions of Parnin and Orso [118]. Specifically, first, instead of locating the faulty statement, we focus on method-level fault localization since existing work has demonstrated that this is more practical in general development [82]. For example, suppose a faulty constructor method misses some field-initialization statements, statement-level fault localization even does not have a ground truth since such field-initialization statement can be added anywhere within the constructor, but method-level fault localization can still work. Second, instead of using percentage, we use Top-N and AWE, two absolute-ranking-based metrics to evaluate the effectiveness of our approaches. However, as Parnin and Orso [118] argued, fault ranking should be more specific under different scenarios, including absolute ranking, search ranking, etc. On the other hand, fault localization techniques should also provide more information to developers to help them understanding the faults instead of just the locations. Honestly, these two points are not fully considered in our approach. In further work, we plan to conduct a user study to further investigate the effectiveness and practicality of our approach in daily development.

2.7 Related Work

To the best of our knowledge, this work is the first extensive study on improving SBFL using PageRank and Method-level Aggregation. We list the related work in fault localization as follows.

2.7.1 Spectrum-Based Fault Localization

Various formulas have been proposed for computing suspiciousness scores of program entities based on passing and failing test cases. Jones et al. [72] proposed the first foundational ranking formula, Tarantula, which is based on the intuition that program entries which are frequently executed by failing test cases and infrequently executed by passing test cases are more likely to be faulty. Dallmeier et al. [41] proposed Ample, an Eclipse plug-in for identifying faulty classes in Java software. Abreu et al. [18] designed Ochiai, which is also widely-studied and state-of-the-art ranking formula. Naish et al. [109] proposed the theoretical-best ranking formulas for single faults, Op and Op2, and empirically analyzed the existing ranking formulas on C programs. Yoo [164] generated a group of ranking formulas using genetic programming (GP). Xie et al. [157] summarized the existing ranking formulas and theoretically compared them, finding that formulas of two families are optimal, including Op, Op2 and four GP-generated formulas. Lucia et al. [98] investigated the effectiveness of the existing work and concluded that there is no best single ranking formula for all cases. Similarly, Steimann et al. [143] studied the threats to the validity in SBFL on ten open-source programs and showed that well-known

fault locators do not uniformly perform better.

2.7.2 Machine Learning-Based Fault Localization

Several existing works have applied machine learning techniques to improve the accuracy of SBFL. Abreu et al. [19] proposed Barinel, in which Bayesian reasoning is used to deduce multiple-fault candidates and their probabilities. Nath et al. [110] proposed TFLMs, a Relational Sum-Product Network model for fault localization. TFLMs can be learned from a corpus of faulty programs and localizes faults in a new context more accurately. Feng et al. [51] proposed Error Flow Graph (EFG), a Bayesian Network to predict fault locations. EFG is constructed from the dynamic dependency graphs of the programs and then standard inference algorithms are employed to compute the probability of each executed statement being faulty. Xuan et al. [159] proposed Multric, which applied RankBoost, a pairwise learning-to-rank algorithm to combine 25 existing formulas. Roychowdhury et al. [130] utilized feature selection for fault localization and Le et al. [85] extended a standard feature selection to identify program entities. Recently, Sohn et al. [142] and Li et al. [94] combined SBFL with source code metrics and mutation information, respectively, for learning-to-rank based fault localization. Perez et al. [123,125] proposed Q-SFL, which leverages qualitative reasoning to augment the information made available to improve SBFL techniques. Li et al. [93] proposed DeepFL, which uses Deep Neural Networks to combine the spectrum-based suspiciousness, mutation-based suspiciousness, complexity-based fault prone-

ness and textual similarity information from the program to boost fault localization accuracy. Actually, our work can be treated as an unsupervised-learning-based fault localization technique. Different from the most existing learning-based work, our work does not require training data which can be hard to collect.

2.7.3 Mutation-Based Fault Localization

Besides spectrum-based and machine-learning-based fault localization, there is one category of approach utilizing mutation analysis [113, 115, 116]. Papadakis et al. [114] firstly applied mutation testing to traditional fault localization. Zhang et al. [168] firstly applied mutation testing to localize faults during regression testing. Later on, Moon et al. [105] proposed MUSE, a mutation-based fault localization technique by analyzing mutant impacts on faulty and correct program entities. Hong et al. [67] developed new mutation operators as well as traditional operators to improve fault localization in real-world multilingual programs. There are also empirical studies evaluating mutation-based fault localization techniques [37, 94, 120]. More specifically, recently, Li et al. [94] demonstrated that mutation-based fault localization can be effective for localizing real-world bugs at the method level.

2.7.4 Slicing-Based Fault Localization

Slicing technique is also widely used in fault localization [66, 175, 176]. Zhang et al. [174] proposed a forward and bidirectional dynamic slicing tech-

niques for improving fault localization. Alves et al. [21] used dynamic slicing technique and change-impact analysis to prune irrelative code statements to improve Tarantula [72]. Sinha et al. [141] focused on the fault localization of Java Runtime Exceptions. They combined dynamic analysis and static backward data-flow analysis to detect source statements which lead to exceptions. Xuan et al. [160] proposed to use program slicing to trim test cases into minimal fractions to achieve more precise SBFL. Gupta et al. [65] combined delta debugging which can identify a minimal failure-inducing input with forward and backward dynamic program slicing to narrow down probably faulty code for improving fault localization. Ocariza et al. [111] also proposed an automated technique to improve fault localization for JavaScript code via backward slicing.

2.7.5 Other Fault Localization Techniques

Similar to mutation-based fault localization, Jeffrey et al. [70] proposed a value-profile-based approach for ranking program statements according to their likelihood of being faulty. Campos et al. [34] applied entropy theory in fitness function to generate new test cases in order to improve fault localization. Alipour et al. [20] extracted extended invariants such as execution features to improve fault localization. Zhang et al. [173] identified the causes of faults by switching predicates' outcome at runtime and altering the control flow. Yu et al. [165] introduced multiple kinds of spectrum types such as control and data dependencies to build fault localization model. Le et al. [86] and

Dao et al. [42] combined SBFL with information-retrieval-based fault localization, which recommends a set of program entities with similar contents of bug reports. Perez et al. [126] proposed a metric, DDU, which applies a quantified test-suite’s diagnosability to SBFL to improve the effectiveness of fault localization. Gonzalez-Sanchez et al. [57] proposed RAPTOR , a test prioritization algorithm to improve fault localization, based on reducing the similarity between statement execution patterns. Le et al. [25] employed likely invariants and suspiciousness scores to locate faults. In this work, they used Daikon’s Invariant Diff [48] tool to mine changes in invariant sets between failing and passing program executions, and applied the learning-to-rank algorithm to predict fault locations.

Chapter 3

DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems¹

The train came out of the long tunnel into the snow country. The earth lay white under the night sky. The train pulled up at a signal stop.

—Yasunari Kawabata, Snow Country

The above quotation is from the first paragraph of fiction “Snow Country”, which describes the scene when the protagonist Shimamura enters the snow country. Back to that time, train was the major vehicle for long-distance travels, while people have more choices today. Now, suppose Shimamura takes a Tesla in Autopilot mode [2], after coming out of the tunnel, there raises a question: can the autopilot system operate safely on the snow-covered road, or the story just ends with a tragedy?

¹This chapter is based on *DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems* [172]. This work is collaborated with Yuqun Zhang, Lingming Zhang, Cong Liu and Sarfraz Khurshid. Mengshi Zhang is the major contributor of this work, who purposed idea, designed experiments and wrote the main sections of the paper.

Autonomous driving is expected to transform auto industry. Typically, autonomous driving refers to utilizing sensors (cameras, Radar, Lidar, GPS, etc) [150] to automatically control vehicles without human intervention. The recent advances in Deep Neural Networks (DNNs) enable autonomous driving systems to adapt their driving behaviors according to dynamic environments [2, 30]. In particular, an end-to-end supervised learning framework is made possible to train a DNN for predicting driving behaviors (e.g., steering angles) by inputting driving images, using $\langle \text{driving image}, \text{driving behavior} \rangle$ pairs as training data. For instance, DAVE-2 [30], released by NVIDIA in 2016, can accurately predict steering angles based on only images captured by a single front-centered camera of autonomous cars.

Recent testing techniques [121, 150] demonstrate that the autonomous driving systems are error-prone to synthetic images of driving scenes. DeepXplore [121] applies differential testing technique to systematically generate images which disclose the inconsistent behaviors of multiple DNN systems. Specifically, it formulates the image generation problem as a joint optimization problem, which uses gradient-based search techniques to find images for maximizing neuron coverage and the number of inconsistent behaviors of such systems. DeepTest [150] designs systematic ways to automatically generate test cases, seeking to mimic real-world driving scenes. Its main methodology is to transform labeled images of driving scenes by applying simple affine transformations and various effect filters such as blurring/fog/rain to the original images, and check if the autonomous driving systems perform consistently



Figure 3.1: Driving Scenes Synthesized by DeepXplore (a)(b) and DeepTest (c)(d)(e)

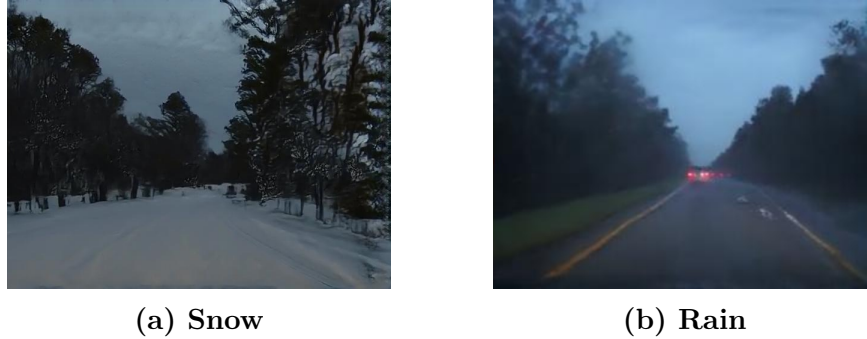


Figure 3.2: Snowy and Rainy Scenes Synthesized by DeepRoad

among the original and transformed scenes. With large amounts of original and transformed driving scenes, DeepTest can detect various erroneous driving behaviors for some well-performed open-source autonomous driving models cheaply and quickly.

However, we observe that the methodologies applied in DeepXplore and DeepTest to generate test cases may not accurately reflect the real-world driving scenes, which can rarely contain colored patch or black holes and sidelines; the blurring/fog/rain effects made by simple simulation also appear to be unrealistic, which compromises their efficacy and reliability. For instance, Figure 3.1 shows the synthetic images which are quoted from the papers of DeepXplore and DeepTest. Note that the colored arrows are attached to present the pre-

dicted steering angles. From Figure 3.1a, 3.1b and 3.1c, it can be observed that the images of driving scenes include several artifacts (patch, holes, and side-lines), which significantly hurt the image quality. Moreover, for Figure 3.1d, it appears to be synthesized by simply dimming the original image and mixing it with the scrambled “smoke” effect and it violates the facts that the density of fog varies along with depth. Similarly, in Figure 3.1e, DeepTest simply simulates rain by adding a group of lines over the original image. This rain effect transformation is even more distorted because usually when it rains, the camera or front windshield tends to be wet and the image is highly possible to be blurred. These facts show that it is difficult to determine whether the erroneous driving behaviors are caused by the flaws of the DNN-based models or the inadequacy of the testing technique itself. Furthermore, these transformations (e.g. translation, shear, and rotation) can only generate similar images, while they cannot sophisticatedly synthesize images with different styles and thus limit the diversity of test cases. For instance, the snowy road condition demands different complicated transformations for rendering the texture of road and roadside objects (such as trees), and it cannot be generated by simple transformation rules.

For traditional software, input validation (IV) is an important step before executing programs. For instance, in web applications, IV checks and filters illegal or malicious inputs to prevent application-level attacks such as buffer overflow and code-injection attack [91]. However, to the best of our knowledge, current DNN-based systems lack to validate inputs (e.g., images

of driving scenes) and thus tends to cause system vulnerability. Specifically, invalid inputs such as outlier images of driving scenes can highly degrade the prediction accuracy and dramatically increase the risks of DNN-based systems. For example, suppose a DNN-based autonomous driving system is trained on a dataset which only includes images of sunny driving scenes. For out-of-domain inputs (e.g. rainy images of driving scenes) that the system is not trained with, it is highly possible that the system outputs wrong control signals which lead to danger for drivers and passengers.

To address the above issues, in this work, we propose an unsupervised learning framework, namely DeepRoad, to systematically analyze DNN-based autonomous driving systems. DeepRoad is composed of a metamorphic testing module, DeepRoad_{MT} and an input validation module, DeepRoad_{IV} . DeepRoad_{MT} employs a Generative Adversarial Network (GAN)-based technique [58, 96] to synthesize driving scenes with various weather conditions, and develops a metamorphic testing module for DNN-based autonomous driving systems. Specifically, the metamorphic relations are defined such that no matter how the driving scenes are synthesized to cope with different weather conditions, the driving behaviors are expected to be consistent with those under the corresponding original driving scenes. At this point, DeepRoad_{MT} enables us to test the accuracy and reliability of DNN-based autonomous driving systems under different scenarios, including heavy snow and hard rain, which can greatly complement the existing approaches (e.g., DeepXplore, DeepTest). For instance, Figure 3.2 presents the snowy and rainy scenes gener-

ated by DeepRoad_{MT} (from sunny scenes), which can hardly be distinguished from genuine ones and cannot be generated using simple transformation rules. DeepRoad_{IV} is designed to validate inputs for DNN-based autonomous driving systems based on image similarity. Firstly, DeepRoad_{IV} applies a pre-trained DNN model-VGGNet to extract high-level features (i.e. contents and styles) of both training and test input images. Then, the Principle Component Analysis (PCA) technique is applied to these features for dimension reduction. Finally, DeepRoad_{IV} validates inputs by comparing the average distance between training and input images with a preset threshold.

To evaluate the effectiveness of DeepRoad, we first synthesize driving scenes under heavy snow and hard rain. In particular, based on GAN, we collect images with two extreme weather conditions from YouTube videos to transform real-world driving scenes and deliver them with the corresponding weather conditions. Subsequently, these synthetic scenes are used to test three open-source DNN-based autonomous driving systems from Udacity community [6]. The experimental results reveal that DeepRoad_{MT} can effectively detect thousands of inconsistent behaviors of different levels for these systems. Furthermore, we use DeepRoad_{IV} to validate the input images sampled from different driving scenes. The results demonstrate that in the embedding space, the cluster of the rainy and snowy image points are separately distributed to the cluster of training images, however, the training cluster is mixed with the majority of the sunny image points. It indicates that given a proper threshold, DeepRoad_{IV} can effectively validate input, which potentially improves the

system robustness.

The key contributions of this work are as follows.

- We propose the first GAN-based metamorphic testing approach to generate driving scenes with various weather conditions for detecting inconsistent behaviors of autonomous driving systems.
- We propose a novel approach to validate inputs for DNN-based autonomous driving system. We present that the distance between the high-level features of training and input images can be used for validating inputs.
- We implement the proposed approaches in DeepRoad, which can generate images of diverse driving scenes (e.g. rain and snow) and measure the similarity between multiple image sets in embedding space. We use DeepRoad to test well-recognized DNN-based autonomous driving models and successfully detect thousands of inconsistent driving behaviors. Additionally, DeepRoad can accurately distinguish images with extreme weather conditions to the training images, which is effective to validate input for autonomous driving systems.

3.1 Background

Autonomous driving systems have been rapidly evolving in recent years [30, 127]. For example, many major auto manufacturers (including Tesla, GM, Volvo, Ford, BMW, Honda, and Daimler) and IT companies (including Waymo/Google, Uber, and Baidu) are working on building and testing various autonomous driving systems. Typically, autonomous driving systems capture data from the environment via multiple sensors (e.g. camera, Radar, Lidar, GPU, IMU, etc.) as input, and use Deep Neural Networks (DNNs) to process data and output control signals (e.g. steering and braking decisions). In NVIDIA’s work [30], their autonomous driving system, DAVE-2 can fluently control cars only based on the images captured by a single front camera. In this work, we mainly focus on DNN-based autonomous driving systems with camera inputs and steering angle outputs.

3.1.1 DNN Architectures

To date, Convolutional Neural Network (CNN) [83] and Recurrent Neural Network (RNN) [132] are the most widely used DNNs for autonomous driving systems. Typically, CNNs are good at analyzing visual imagery and RNNs can effectively process sequential data. In this work, the evaluated models are built on CNN and RNN modules. We briefly introduce the basic concepts and components of each architecture as follows, where more details about DNNs are provided in [88].

3.1.1.1 Convolutional Neural Networks

Convolutional Neural Networks are similar to regular neural networks, which include a large number of neurons and pass information in a feed-forward way. However, since the input data are images, several properties can be applied to optimize the regular neural networks, where the convolutional layer is a key component in CNNs. Instead of being fully connected, a neuron in a layer only connects to some neurons in the previous layer, and the computational process can be presented as a convolution with kernels. Figure 3.3a shows an example of CNN-based autonomous driving system that consists of an input layer (images) and an output layer (steering angles), as well as multiple hidden layers. Convolution hidden layers allow weight sharing across multiple connections and can greatly save training efforts.

3.1.1.2 Recurrent Neural Networks

Regular neural networks and CNNs are designed to process independent data, such as using CNN to classify images. However, for sequential data like videos, the neural networks should not only capture information of each single frame, but are also expected to model the connections between them. Unlike regular NNs and CNNs, RNN is a kind of neural network with feedback connections. As shown in the left part of Figure 3.3b, RNNs use loops to forward the previous states to input, which model the connection of input data. The right part of Figure 3.3b shows the workflow of the unfolded RNN for predicting steering angles based on a sequence of images. At each step,

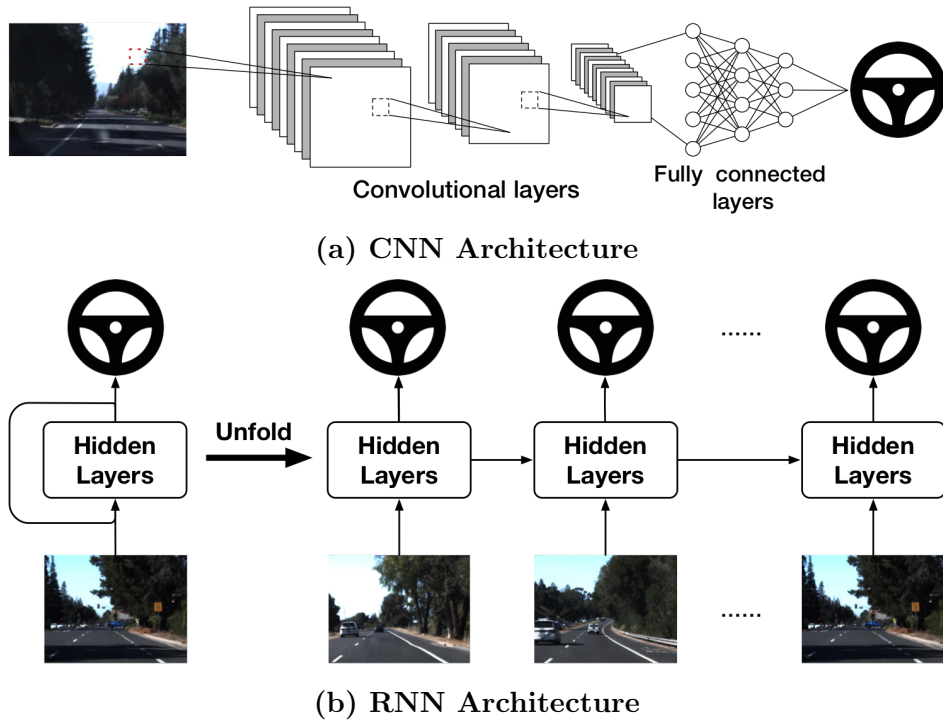


Figure 3.3: Autonomous Driving Systems Built on CNN and RNN RNN takes the current input image and previous hidden states as input and predicts the steering angle.

DNN-based autonomous driving systems are essentially software systems, which are error-prone and can lead to tragedies. For example, in January 2018, a Tesla Model S plowed into a fire truck at 65 mph while using Autopilot [16]. And in Mar 2018, an autonomous Uber failed to slow down and killed a pedestrian during road test at night [17]. To ensure the quality of software systems, many software testing techniques have been proposed in the literature [22, 102], where typically, a set of specific test cases are generated to test if the software programs perform as expected. The process of deter-

mining whether the software program performs as expected upon the given test inputs is known as the *test oracle* problem [22]. Despite the abundance of traditional software testing techniques, they cannot be directly applied for DNN-based systems since the logics of DNN-based software is learned from data with minimal human interference (like a blackbox) while the logics of traditional software programs are manually created.

3.2 Approach

3.2.1 Metamorphic Testing for DNN-Based Autonomous Driving Systems

3.2.1.1 Metamorphic DNN Testing

Metamorphic Testing [134] (MT) has been widely used to automatically generate tests to detect software faults. The strength of MT lies in its capability to automatically solve the test oracle problem via Metamorphic Relations (MRs). In particular, let p be a program mathematical representation that maps program inputs to program outputs (e.g., $p[i] = o$). Assuming f_I and f_O are two specific functions for transforming the input and output domain respectively, and they satisfy the following MR formulation:

$$\forall i, p[f_I(i)] = f_O(p[i]) \quad (3.1)$$

, where i denotes the input of program p .

With such MRs, we can test a specific implementation $\hat{p}$ of p by checking whether $\hat{p}[f_I(i)] = f_O(\hat{p}[i])$ for various input i . Accordingly, MT is defined as testing a program implementation via cross-checking inputs and outputs with MRs. For instance, given a program implementing function *sine*, MT can be used to delineate test oracles and create various new tests. For any existing input i to test function *sine*, various facts can serve as MRs, e.g., $\sin(-i) = -\sin(i)$ and $\sin(i + 2\pi) = \sin(i)$. These facts can be formulated as 1) $f_I(x) = f_O(x) = -x$, 2) $f_I(x) = x + 2\pi$ and $f_O(x) = x$. With such MRs, we can transform the existing test inputs according to f_I to generate additional tests, and check the output based on f_O . For instance, suppose the default test

case of function *sine* is `AssertTrue(sin(0.5·Pi), 1.0)`. Based on the above MRs, we can generate two extra tests: `AssertTrue(sin(-0.5·Pi), -1.0)` and `AssertTrue(sin(2.5·Pi), 1.0)`.

In this work, we further apply MT to test DNN-based autonomous driving systems. Formally, denote *DNN* as a DNN-based autonomous driving system that continuously maps each image into predicted steering angle signal (e.g., turning left for 15°). One MR can be defined as given the original image stream $\mathbb{I}$, various image transformations $\mathbb{T}$ can simply change the road scenes (detailed shown in Section 3.2.1.2) without impacting the predictions for each image $i \in \mathbb{I}$ (e.g., the predicted direction should be approximately the same on the same road under different weather conditions). This MR to test DNN with additional transformed inputs can be formalized as follows:

$$\forall i \in \mathbb{I} \wedge \forall \tau \in \mathbb{T}, DNN[\tau(i)] = DNN[i] \quad (3.2)$$

3.2.1.2 DNN-Based Road Scene Transformation

The recent work DeepTest [150] also applied MT to test DNN-based autonomous driving systems. However, it only performs basic image transformations, such as adding simple blurring/fog/rain effect filters, and thus has the following limitations: (1) DeepTest may generate images which violate common scenes (discussed in Section 3). (2) DeepTest cannot simulate complex road scene transformations (e.g., snowy scenes).

To complement DeepTest by automatically generating various real-world road scenes, in this work, we leverage UNIT [96], a recent published

DNN-based method to perform unsupervised image-to-image transformation based on Generative Adversarial Networks (GANs) [58] and Variational Autoencoders (VAEs) [80]. One insight of UNIT is that suppose two images contain the same contents but lie in different domains, they should have the same representations in a shared-latent space. Accordingly, given a new image from one domain (e.g., the original driving scene), UNIT can automatically generate its corresponding version in the other domain (e.g., rainy driving scene).

Figure 3.4 [96] presents the structure of UNIT, S_1 and S_2 denote two different domains (e.g., sunny and rainy driving scenes), E_1 and E_2 denote two autoencoders which project the images from S_1 and S_2 to a shared-latent space Z . Suppose x_1 and x_2 are paired images which share the same content. Ideally, E_1 and E_2 would encode them to the same latent vector z , and it can be translated back to S_1 and S_2 by two domain-specific generators G_1 and G_2 , respectively. D_1 and D_2 are two discriminators which detect whether the image belongs to S_1 and S_2 respectively. Specifically, they are expected to differentiate whether the input image is sampled from the target domain (e.g. a real image) or produced by a well-trained generator (e.g. a synthetic image). Based on the autoencoders and generators, UNIT can be used to transform images between two domains. For instance, image x_1 can be transformed to S_2 by $G_2(E_1(x_1))$.

In UNIT, all D_i , E_i and G_i are incarnated as neural networks, and the learning objective of UNIT can be decomposed to optimize the following costs:

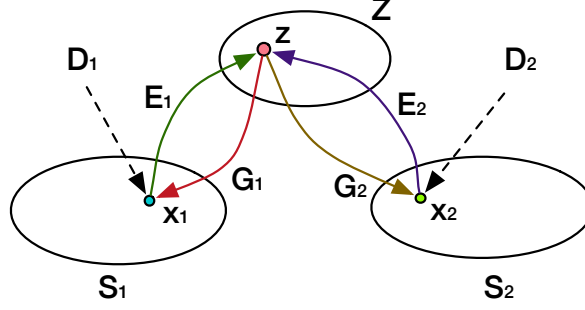


Figure 3.4: Structure of UNIT

- **VAE loss:** minimizing the loss of image reconstruction for each $\langle E_i, G_i \rangle$.
- **GAN loss:** achieving the equilibrium point in the minimax game for each $\langle G_i, D_i \rangle$, where D_i aims at discriminating the images to find out whether they are sampled from the domain S_i or produced by G_i that aims at fooling D_i .
- **Cycle-consistency loss:** minimizing the loss of cycle- reconstruction for each $\langle E_i, G_j, E_j, G_i \rangle$, where x_1 is expected to equal to $G_1(E_2(G_2(E_1(x_1))))$ and x_2 is expected to equal to $G_2(E_1(G_1(E_2(x_2))))$.

The total loss can be summarized as follows:

$$\begin{aligned}
& \min_{E_1, E_2, G_1, G_2} \max_{D_1, D_2} \mathcal{L}_{CC_1}(E_1, G_2, E_2, G_1) \\
& \quad + \mathcal{L}_{CC_2}(E_2, G_1, E_1, G_2) \\
& \quad + \mathcal{L}_{VAE_1}(E_1, G_1) + \mathcal{L}_{VAE_2}(E_2, G_2) \\
& \quad + \mathcal{L}_{GAN_1}(D_1, G_1) + \mathcal{L}_{GAN_2}(D_2, G_2)
\end{aligned}$$

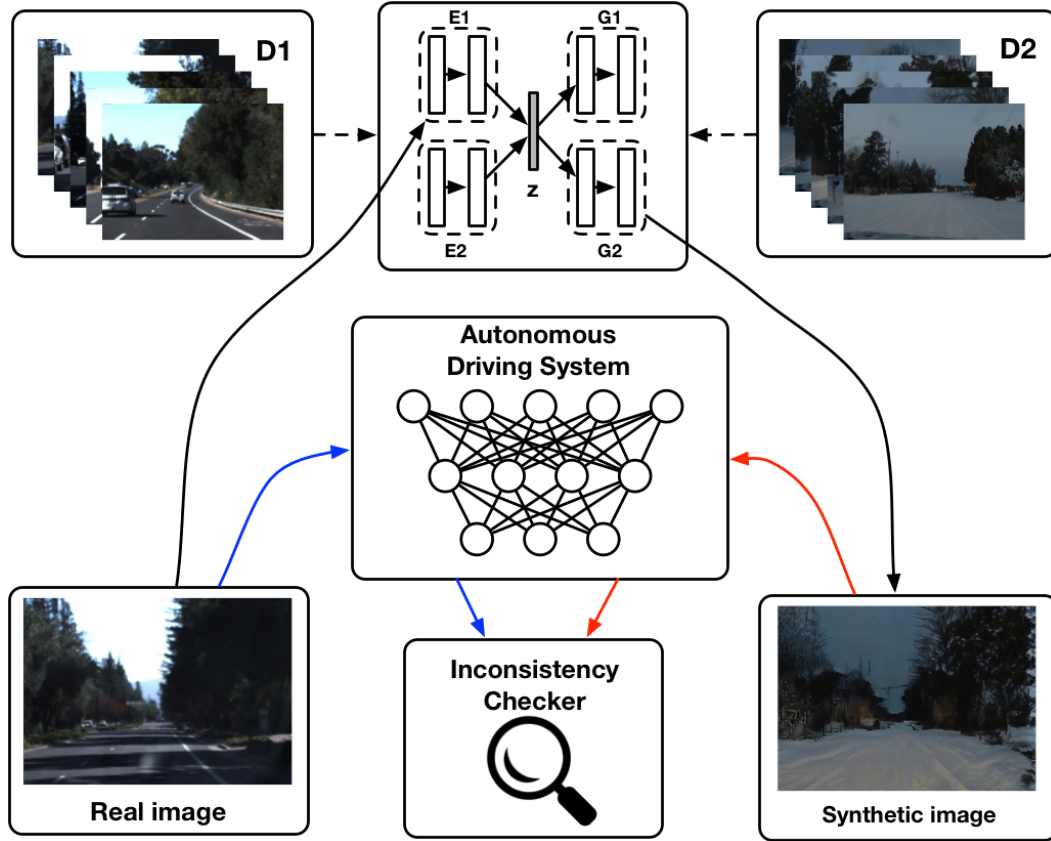


Figure 3.5: Framework of DeepRoad_{MT}

, and this loss function can be optimized using Stochastic Gradient Descent algorithm.

3.2.1.3 Framework of DeepRoad_{MT}

Figure 3.5 shows the overall design of our metamorphic testing framework for DNN-based autonomous driving systems—DeepRoad_{MT}. In Figure 3.5, DeepRoad_{MT} first takes *unpaired* training images from two target domains (e.g., datasets of the driving scene under sunny and snowy weather respec-

tively), and utilizes UNIT to project two domains to the same latent space by optimizing the loss functions presented in Section 3.2.1.2. When the training process finished, DeepRoad_{MT} uses the well-trained model to transform the whole dataset of sunny driving scenes to snowy weather. Specifically, given any image under sunny weather i , DeepRoad_{MT} encodes it to vector z_i by E_1 , and synthesizes its corresponding version under snowy weather $\tau(i)$ using G_2 . DeepRoad_{MT} feeds each pair of real and synthetic driving scene images to the autonomous driving systems under test, i.e., DNN , and compare their prediction results $DNN[\tau(i)]$ and $DNN[i]$ to detect any inconsistent behaviors. Normally, the transformed driving scenes are expected not to significantly impact the predicted steering angles, and any inconsistency may indicate correctness or robustness issues of the systems under test [121, 150].

3.2.2 Input Validation for DNN-Based Autonomous Driving Systems

Driving scenes synthesized by DeepTest and DeepXplore can be used as test cases to test DNN-based autonomous driving systems in an offline manner. Though these test cases are useful to expose the system vulnerability and advise developers to complement training data from real world to improve the system robustness, it is not sufficient for online testing. For instance, a DNN-based autonomous driving system can be well trained and perfectly function in sunny environments, yet it might perform incorrectly at night or on a snow-covered road, because the lane marks it detected for guiding cars disappear in such driving scenes. This example suggests that if the system can validate

input images online, and actively advise drivers to control the car when it cannot handle the invalid inputs, the autonomous driving systems can become safer and more robust. In the following, we first define the criteria of input validation for DNNs (especially image-oriented models) and present our input validation framework for DNN-based autonomous driving systems.

3.2.2.1 Input Validation of DNNs

The goal of input validation (IV) is to ensure that only properly formed data can be accepted by systems, and malformed data should be rejected before execution. The reason is that an invalid input may trigger malfunction of downstream components, which makes the system insecure. Generally, the valid input of a program can be explicitly defined such as the input string should not be null/empty or the value of a certain input variable should be greater than 0. However, it is not trivial to properly define the input validity of a DNN-based program. For example, we can define an IV criteria as the input data should be any RGB image with size 640*480, or any input data should exist in the training dataset to guarantee the correctness. However, none of them are proper since the first criteria is too weak to improve system robustness, and the second is so strong that makes the system lack generalisability.

We define the IV criteria of DNN-based program based on the Probably Approximately Correct (PAC) Learning theory. According to the PAC Learning theory [29], a machine learning model Λ is expected to learn the distribution $\mathcal{D}$ from the training dataset, and predict the correct label with high

probability.² This can be formulated as follows:

$$E(\Lambda; \mathcal{D}) = Pr_{\mathbf{x} \sim \mathcal{D}}(\Lambda(\mathbf{x}) \neq y) \quad (3.3)$$

$$Pr(E \leq \epsilon) \geq 1 - \sigma \quad (3.4)$$

In Formula 3.3, E denotes that the probability of Λ makes an incorrect prediction ($\Lambda(\mathbf{x}) \neq y$) on input data $\mathbf{x}$ sampled from $\mathcal{D}$, and in Formula 3.4, ϵ and σ are two parameters between 0 and 1, such that it is highly possible (greater than $1 - \sigma$) E is small (less than ϵ), which means Λ is effective on $\mathcal{D}$. Based on the above equations, we first define an abstract IV criterion of DNN-based systems is that the input data should be sampled from $\mathcal{D}$. As discussed before, suppose the input data is not sampled from $\mathcal{D}$, the IV criterion is violated and the prediction accuracy is not guaranteed. Therefore, it is necessary to validate inputs to improve the robustness of DNN-based systems.

Intuitively, the IV criterion should be instantiated as:

$$Pr_{\mathbf{x} \sim \mathcal{D}}(\mathbf{x} = i) > \theta \quad (3.5)$$

, which means the probability of input i being sampled from $\mathcal{D}$ should be greater than the predefined threshold θ . Otherwise, the system refuses to predict on i . However, this definition is not tractable for image data because image data is highly dimensional and their distribution (e.g. Gaussian Mixture model) is difficult to be explicitly represented. To address this issue, we

²For simplicity, here we only discuss DNNs for classification, the same approach can be applied to explain DNNs for other tasks such as regression.

project image data to a low-dimensional space and use the distance between inputs and training data to replace $Pr_{\mathbf{x} \sim \mathcal{D}}(\mathbf{x} = i)$. In particular, according to the Manifold Learning theory [29], the images generated by $\mathcal{D}$ can be embedded into a non-linear low-dimension manifold $M_{\mathcal{D}}$. Suppose the input data i is sampled from $\mathcal{D}$, its projection i_p should be included by $M_{\mathcal{D}}$. Furthermore, we propose an extra constraint for the non-linear embedding that suppose the input data are generated by a different distribution $\mathcal{D}'$, their projections are expected to be included by another manifold $M_{\mathcal{D}'}$, which is linearly separable to $M_{\mathcal{D}}$. Based on the constraint, we can compute the minimal distance of i_p and the projections of training data to validate if i_p belongs to $M_{\mathcal{D}}$. The IV criterion is redefined as follows:

$$\min_j \|h(i) - h(j)\|_2 < \theta' \quad (3.6)$$

, where $\|\cdot\|_2$ denotes L_2 norm, $h(\cdot)$ denotes the required non-linear projection and θ' denotes predefined threshold for input validation. If the input satisfies Equation 3.6, it will be processed by DNNs for prediction, otherwise, it will be rejected.

3.2.2.2 Framework of DeepRoad_{IV}

We propose DeepRoad_{IV}, an input validation framework for autonomous driving systems. DeepRoad_{IV} separate the projection $h(\cdot)$ to two parts: non-linear transformation and dimension reduction. For the first part, DeepRoad_{IV} applies VGGNet [140], a widely used DNN [53,71] to extract high-level features from each image. To be specific, the input image is encoded in each layer of

VGGNet by kernels. Suppose layer i includes N_i distinct kernels, it generates N_i feature maps each of size (w_i, h_i) , where w_i and h_i are the width and height of the feature maps respectively. These feature maps can be stored as a feature matrix F_i with size (N_i, M_i) , where each row of F_i is the vector flattened from the corresponding feature map and M_i is $w_i \cdot h_i$. DeepRoad_{IV} also generates style information which is introduced in [52]. This style information aims at capturing the texture of images and it is defined by feature correlation, which can be computed by the Gram matrix

$$G_i = F_i \cdot F_i^T \quad (3.7)$$

Suppose we choose layer i and j to extract the feature and style matrix F_i and G_j respectively, the representation vector of the given image is $\vec{v} = [v_F, v_G]$, where v_F , v_G are the flattened vector of F_i and G_j respectively. Further, we apply Principle Component Analysis (PCA) technique to reduce feature dimension of input and training data as follows:

$$Y = X \cdot P \quad (3.8)$$

X denotes the input matrix with size (n, m) , where n is the total number of input and training data and m is the length of feature vector $\vec{v}$. P denotes the projection matrix with size (m, k) , where k is the target dimension less than m , and P can be computed using X [29].

Figure 3.6 shows the overall design of our input validation framework, DeepRoad_{IV}. DeepRoad_{IV} first takes the training and online driving images as

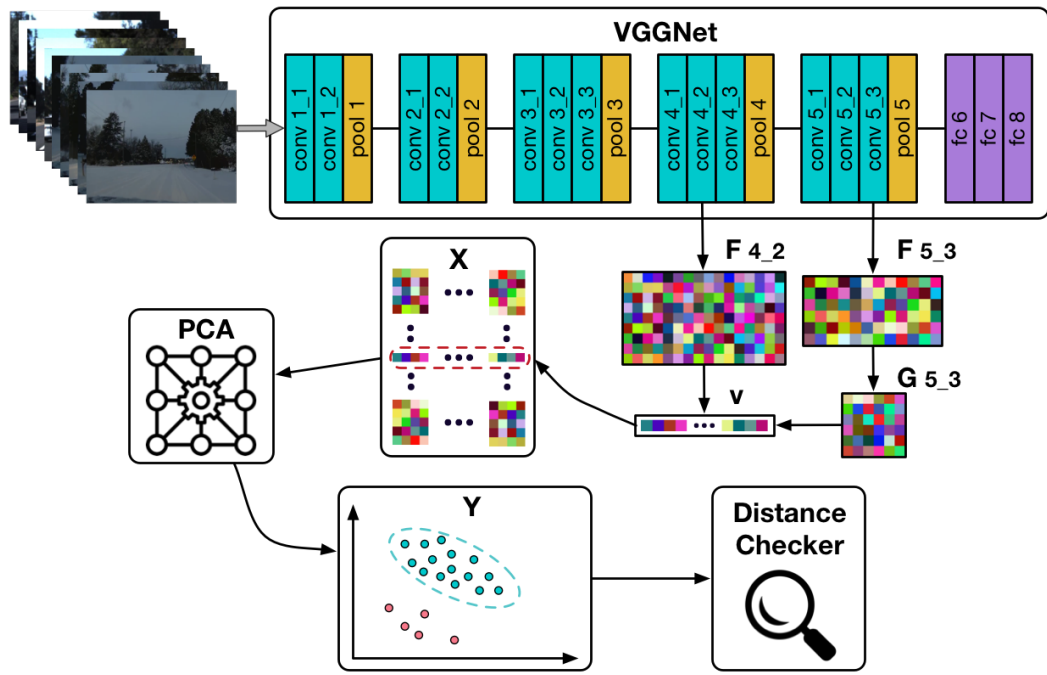


Figure 3.6: Framework of DeepRoad_{IV}

input, and uses VGGNet to extract their content and style features. As shown in Figure 3.6, DeepRoad_{IV} inputs a snowy image to VGGNet, and chooses the convolutional layer conv 4_2 and conv 5_3 to extract content and style features respectively. To be specific, the colored grids $\mathbf{F}_{4_2}$ and $\mathbf{F}_{5_3}$ denote the content features extracted from VGGNet, and the style feature $\mathbf{G}_{5_3}$ is computed by Equation 3.7. Note that these colored grids are just used to visualize results, and their dimensions do not match the real outputs. Then, matrix $\mathbf{F}_{4_2}$ and $\mathbf{G}_{5_3}$ are flattened and concatenated to feature vector $\mathbf{v}$. DeepRoad_{IV} processes all image data using the same approach and the feature vectors compose matrix $\mathbf{X}$. In the second step, DeepRoad_{IV} applies PCA to reduce the feature dimension. In Figure 3.6, we set the target dimension to 2. The processed data $\mathbf{Y}$ are presented on a 2-D plane, where the blue and red nodes denote the training and online driving images respectively. Finally, DeepRoad_{IV} computes the minimal distance between training data and each online image and refuses to predict for the images whose distances are greater than a certain threshold.

Dataset	Frame	Duration	Weather Cond.
Udacity Training	33805	N.A.	Sunshine
Udacity Test Ep1	15212	N.A.	Sunshine
Udacity Test Ep2	5614	N.A.	Sunshine
Youtube Ep1	1000	28:55	Heavy snow
Youtube Ep2	1000	1:09:03	Hard rain

Table 3.1: Details of image sets

3.3 Experiments

3.3.1 Data

We use a real-world dataset released by Udacity [7] as a baseline to check the inconsistency of autonomous driving systems. From the dataset, we select two episodes of the high-way driving video where obvious changes of lighting and road conditions can be observed among frames. To train UNIT model, we also collect images of extreme scenarios from YouTube. In the experiments, we select heavy snow and hard rain, two extreme weather conditions to transform real-world driving scenes. To make the variance of collected images relatively large, we only search for videos which is longer than 20 mins. In the scenario of hard rain, the video records wipers swiping windshield, which would potentially degrade the quality of synthetic images. Hence, in the data preprocessing phase, we manually check and filter those images. Note that all images used in the experiments are cropped and resized to 320*240, and we have performed down-sampling for YouTube videos to skip consecutive frames with close contents. The detailed information is present in Table 3.1.



(a) Heavy Snow



(b) Hard Rain

Figure 3.7: Images Collected from YouTube

3.3.2 Models

We evaluate our metamorphic testing framework DeepRoad_{MT} on three DNN-based autonomous driving models which are released by Udacity [7]: Autumn [3], Chauffeur [4], and Rwrightman [5]. We choose these three models as their pre-trained models are public and can be evaluated directly on the synthetic datasets. To be specific, the model details of Rwrightman are not



Figure 3.8: Real and GAN-Generated Images

publicly available, however, similar to black-box testing, our approach aims at detecting the inconsistencies of the model. Hence, **Rwrightman** is still used for evaluations.

Autumn. **Autumn** is composed of a data preprocessing module and a CNN. Specifically, **Autumn** first computes the optical flow of raw images and inputs them to a CNN to predict steering angles. The architecture of **Autumn** is: three 5x5 conv layers with stride 2 plus two 3x3 conv layers and followed by five fully-connected layers with dropout. The model is implemented by OpenCV, Tensorflow and Keras.

Chauffeur. **Chauffeur** consists of one CNN and one RNN with LSTM module. The workflow is that CNN first extracts the features of input images and then utilizes RNN to predict the steering angle from the previous 100 consecutive images. This model is also implemented by Tensorflow and Keras.

3.3.3 Metric

Metric of model inconsistency. In this work, an autonomous driving system is defined to act consistently if its steering angle prediction falls within certain error bounds after modifying the weather condition of driving images. We define the number of inconsistent behaviors of autonomous driving systems as follows:

$$IB(DNN, \mathbb{I}) = \sum_{i \in \mathbb{I}} f(|DNN[i] - DNN[\tau(i)]| > \epsilon)$$

, where DNN denotes the autonomous driving model and $\mathbb{I}$ is the real-world driving dataset. i denotes the i th image in $\mathbb{I}$. τ denotes the image generator/-transformer which can change the weather condition of the input image. f is an indicator function which outputs 1 or 0 if and only if the input is *True* or *False* and ϵ is the error bound.

Metric of input validation. As introduced in Section 3.2.2.2, the input validity of DNN-based autonomous driving systems is defined by the minimal distance of input and training images in the embedding space. This metric can reflect the similarity between the input and training data, however, it has the following limitations: first, generally, the training dataset is large (e.g. 10k images). Suppose we use the above metric to validate a single input image, the numerous training data points will dominate PCA and the results are biased. Second, using the minimal distance for input validation is not stable. For example, suppose the distance of input i and training data j is minimal and it is less than the threshold. However, j is far from other training data and actually, i is not similar to the majority of the training dataset. We address these limitations in the following ways: first, to balance the input data and training data, we collect M images from online driving scenes as input data and randomly select M images from training dataset as training data. Second, to estimate the distance more stable, we average the Top-N minimal distances of each image to represent their similarities. The metric of input validity is

defined as follow:

$$m_{IV}(i, S_t) = f(\frac{1}{N} \sum_{k \in \{N\}} \min^k(\|h(i) - h(j)\|_2) < \theta)$$

, where N is a parameter which is less than M , i denotes the image of the input dataset with size M . S_t denotes the set of M randomly selected training images, $\min^k(\cdot)$ denotes the k -th minimal value among input array. Function f is an indicator function and θ is the threshold of input validation.

3.3.4 Results

3.3.4.1 Results of DeepRoad_{MT}

We first present several YouTube screenshots as ground truth in Figure 3.7 to help readers check the quality of synthetic images. In Figure 3.8, we list real and GAN-generated images pairs, where the two rows present the transformation of Udacity dataset to snowy and rainy scenes, respectively, and the odd and even columns present original and GAN-generated images, respectively. Qualitatively, the GAN-generated images are visually similar to the images collected from YouTube videos and they also can keep the major semantic information (such as the shape of tree and road) of the original images. Interestingly, in the first snowy image in Figure 3.8, the sky is relatively dark and GAN can successfully render the snow texture and the light in front of the car. In the second column, the sharpness of rainy images are relatively low and this is consistent with the real scene shown in Figure 3.7. Our results are consistent with the original UNIT work [96], and further demonstrate the effectiveness of UNIT for image transformation.

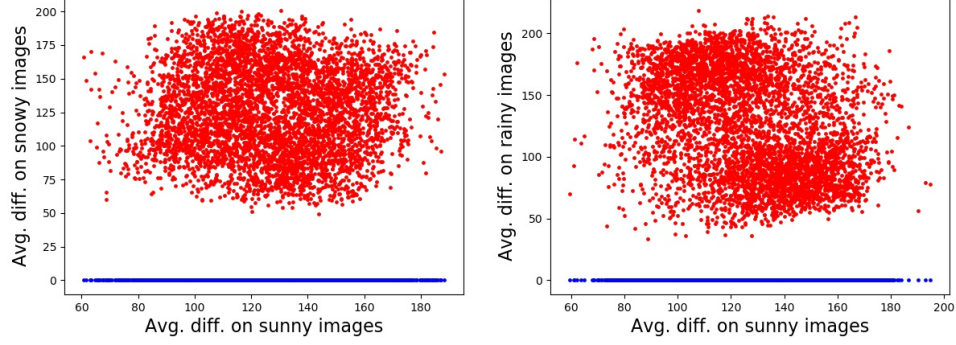


Figure 3.9: Results of Testing Mode Collapse of UNIT Models

Furthermore, we quantitatively study if UNIT can avoid mode collapse such that the model can generate different images from different inputs. We test UNIT as follows: we first randomly sample 5000 pairs of the input images (e.g. sunny images) to compute their average pixel difference. Then, we use the UNIT model to generate the pairs of synthetic images (e.g. snowy images) and compute the average pixel differences as well. We present the experimental results of sunny-to-snowy and sunny-to-rainy models in Figure 3.9. In the figure, the x- and y-axis present the average pixel differences on the input and synthetic image pairs (red spots), respectively. Moreover, we provide a baseline (blue spots) for comparison that the model suffers strong mode collapse such that it can only generate the same image for arbitrary inputs. In the figure, we can observe that there are big gaps between the red clusters and the blue baselines, which means the mode collapse does not happen and the models can generate various images given different inputs.

We further present examples for the detected inconsistent behaviors of



Figure 3.10: Inconsistency of Steering Angle Prediction on Original and Synthetic Images

autonomous driving models in Figure 3.10. In the figure, each figure pair shows the original and associated synthetic mages for testing deep learning models. In each sub-figure, the red caption indicates the predicted steering angles, and the curves visualize the predictions which help check the differences. From the figure we can observe that model **Autumn** has the highest inconsistency number on both scenes; in contrast, model **Rwrightman** is the most stable model under different scenes. This figure shows that DeepRoad_{MT} is able to find inconsistent behaviors under different road scenes for real-world autonomous driving models.

Table 3.2 presents the detailed number of detected inconsistent behaviors under different weather conditions and error bounds for each studied autonomous driving model on the Udacity dataset. For example, when using the error bound of 10° and the rainy scenes, DeepRoad_{MT} detects 5279, 710, and 656 inconsistent behaviors for **Autumn**, **Chauffeur**, and **Rwrightman**, respectively. From the table, we can observe that the inconsistency number of **Autumn** is the highest under both weather conditions. We think one potential reason is that **Autumn** is purely based on CNN, and does not utilize history information (e.g., via RNN), and thus may not always perform well in all road scenes. On the other hand, **Rwrightman** performs the most consistently than the other two models under all error bounds. This result presents a very interesting phenomenon – DeepRoad_{MT} can not only detect thousands of inconsistent behaviors of the studied autonomous driving systems but can also measure different autonomous systems in terms of their robustness. For example, with

Scene	Model	Num. of Inconsist. Behav.			
		10°	20°	30°	40°
Snowy	Autumn	11635	11602	11388	10239
	Chauffeur	4839	2105	1093	653
	Rwrightman	334	115	45	14
Rainy	Autumn	5279	5279	5279	5279
	Chauffeur	710	175	94	71
	Rwrightman	656	92	23	0

Table 3.2: Number of inconsistency behavior of three models under different weather conditions

the original Udacity dataset, it is hard to find the limitations of autonomous driving systems like *Autumn*.

3.3.4.2 Results of DeepRoad_{IV}

We use sunny, rainy and snowy driving scenes to test DeepRoad_{IV}. The expectation of this experiment is, in the embedding space, the sunny images are close to the training images, and the rainy and snowy images are linearly separable to them. Specifically, sunny images are collected from the original test dataset, and the rainy and snowy images are extracted from YouTube videos. Note that to ensure the authenticity of input images, we only choose real-world instead of synthetic images. Moreover, we choose convolutional layer conv 3_2 and conv 4_1 of VGGNet to extract the content and style features from the input images, and we set the PCA dimension to 3 for visualizing experimental results. To reduce the computational complexity, we resize all the images to 120*90 and set the sampling number M of each dataset to 600. Furthermore, we use the average of Top-100 minimal distances of each data

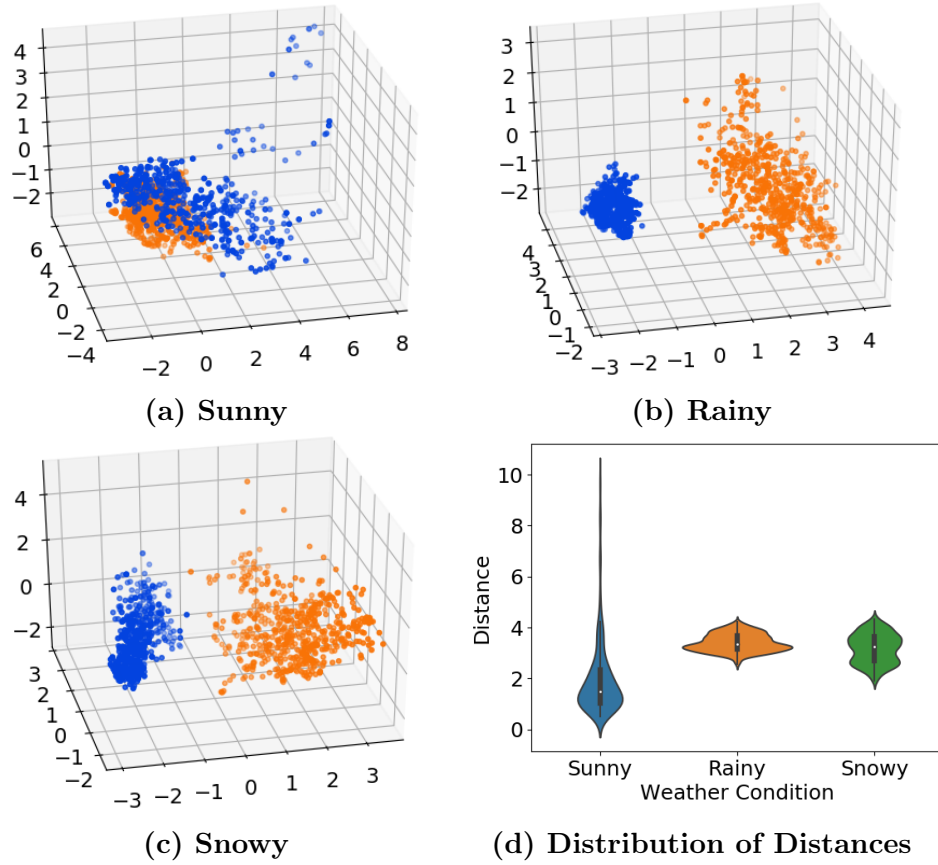


Figure 3.11: Results of DeepRoad_{IV}: Image Embeddings and Distance Distributions

point to reduce the variance of similarity estimation for each input image. Figure 3.11 visualizes the results of DeepRoad_{IV} on sunny, rainy and snowy driving scenes. To be specific, the first three figures of Figure 3.11 present the results of sunny, rainy and snowy images, respectively. And the orange and blue points present the sampled training and corresponding input images. We first analyze the results of the image embedding. From Figure 3.11a, we observe that the majority of the input images are mixed with the training samples, and a few inputs are far from the cluster. From Figure 3.11b and 3.11c, there

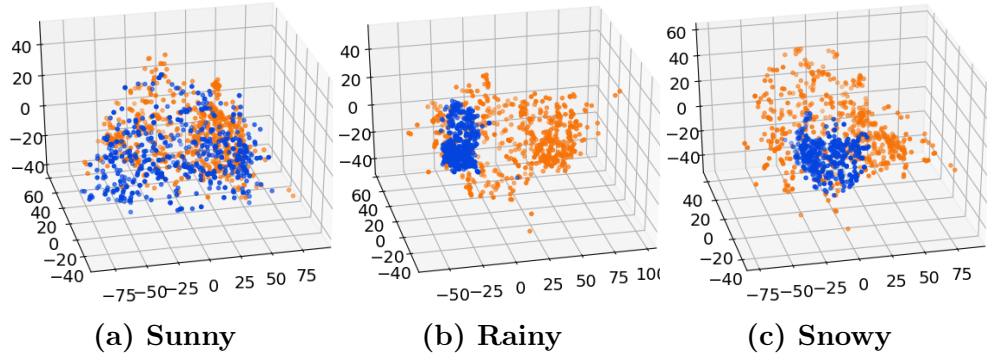


Figure 3.12: Results of DeepRoad_{IV} (without non-linear Transformation): Image Embeddings and Distance Distributions

are gaps between the input and training points and the clusters are linearly separable. These results indicate that the distributions of sunny and training images are close but the rainy and snowy images are not. On the other hand, the cluster of rainy and snowy images are relatively compact but the sunny images are scattered. The reason may be the texture of rainy and snowy images are unified and the content is relatively poor so that the distances between images are small. However, the light condition and content of sunny images are more diverse, hence the distances are large. Moreover, from Figure 3.11d, we find the distances of sunny images mainly lie between 0 and 3, and almost all of the distances of rainy and snowy images are larger than 2. Suppose the threshold of input validation is 2.5, DeepRoad_{IV} can detect 100% of rainy, 85% of snowy images and 21% outliers among sunny images as invalid inputs, which effectively improve the system robustness. Furthermore, we study if the non-linear transformation of input images is necessary for input validation. Figure 3.12 visualizes the results of DeepRoad_{IV} without feature extraction. From Figure 3.12, we observe that all blue clusters are surrounded by the

orange points, which show that input images are not linearly separable to the training images in the embedding space. It implies in this case, the distance is not a proper metric for input validation, and non-linear transformation (i.e. feature extraction using VGGNet) is indeed needed.

3.4 Threats to Validity

There are several threats to the validity of the proposed approach and its result, which include the followings.

In this work, the main threat to internal validity is potential defects in the implementation of our techniques. To reduce these threats, in implementing DeepRoad_{MT}, we used the original implementation of UNIT to ensure DeepRoad_{MT}'s performance. Furthermore, in implementing of DeepRoad_{IV}, we downloaded the pre-trained VGGNet weights from PyTorch website³ instead of training it on ImageNet.

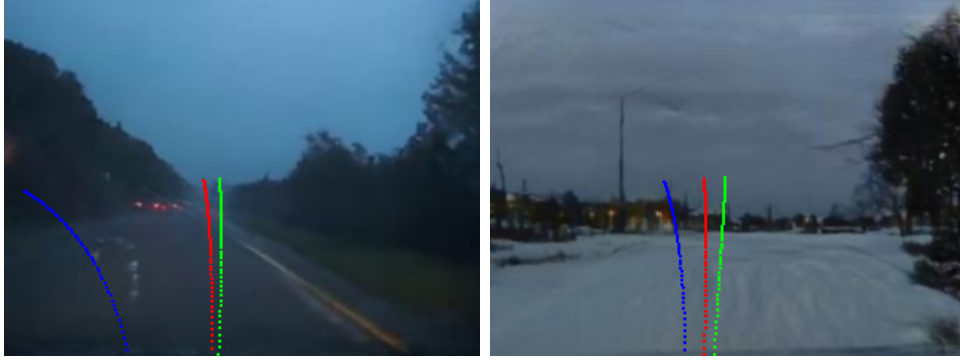


Figure 3.13: Results of Differential Testing

The threats to external validity mainly lie in image quality, dataset, and autonomous driving models. First, we lack a good standard to evaluate image quality (i.e. realism). In this work, we present GAN-generated images to let readers check their quality. This approach is quite straightforward but less objective. Salimans et.al [133] proposed Inception Score to evaluate the quality

³<https://download.pytorch.org/models/vgg19-dcbb9e9d.pth>

of synthetic images. To be specific, Inception Score uses an Inception-v3 Network pre-trained on ImageNet to compute a statistic of the network’s outputs as the quality of generated images. However, Barratt et.al [26] demonstrate that Inception Score fails to provide useful guidance when comparing generative models (e.g. GANs). Furthermore, the generation process of GANs is not controllable and some semantic contents (e.g. lane boundaries) of the synthetic images may be missing or not match the original images, and they threaten the validity of Metamorphic Testing. Here, we provide two solutions to reduce the risks: 1) given a model, DeepRoad_{MT} does not only provide users the statistics of the inconsistent predictions, but it also recommends them to manually check the original and synthetic images. Specifically, the inconsistency can be triggered due to the ADS model cannot handle the domain shift (e.g. weather change) and makes less accurate predictions, or, the synthetic images do not match the original images and associated ground truth is changed. Therefore, the comparison of the original and synthetic images can help users find the root cause of the inconsistency, however, it is expensive and time-consuming since the users are involved in the testing loop. 2) suppose there are multiple models need to test, we can apply Differential Testing [122, 150] technique to study the inconsistent predictions of models on the same image, and it can help avoid the issue of mismatch between the original and synthetic images. The example of Differential Testing on our models and synthetic images are shown in Figure 3.13, where the red, green and blue curves visualize the predictions of model Autumn, Chauffeur and Rwrightman, respectively. Second,

the Udacity dataset is relatively small and the autonomous driving models are quite simple. Suppose the dataset is sufficiently large, a more complicated and robust model is able to be trained, and the inconsistent behaviors would be dramatically reduced. Moreover, an autonomous driving system is complicated, and its input and output are diverse. In this work, we only focus on testing the accuracy of the steering angle instead of speeding adjustments.

3.5 Related work

3.5.1 Metamorphic Testing

Metamorphic testing is a classical software testing method that identifies software bugs [38, 135, 177]. Its key idea is to detect violations of domain-specific metamorphic relations defined across outputs from multiple runs of the program with different inputs. Metamorphic testing has been applied for testing machine learning classifiers [107, 155, 156]. In this work, DeepRoad develops a specific GAN-based metamorphic testing module for DNN-based autonomous driving systems, where the metamorphic relations are defined such that regardless of how the driving scenes are synthesized to cope with weather conditions, the driving behaviors are expected to be consistent with those under the corresponding original driving scenes.

3.5.2 Input Validation

Input Validation aims at ensuring that only properly formed data can be accepted by an information system, and preventing malformed data leading systems errors. Input Validation has been applied to enhance the robustness of web application [1, 91]. In this work, DeepRoad develops a distance-based input validation framework for DNN-based autonomous driving systems, where the key idea is that a valid input image is similar to a part of the images in the training dataset and the similarity can be measured by the distance in a non-linear low-dimension space. To enhance the systems' security, the images will be rejected if their distance is greater than a given threshold.

3.5.3 Testing and Verification of DNN-Based Autonomous Driving Systems

Different from traditional testing practices for DNN models [99, 152], a recent set of approaches (such as DeepXplore [121] and DeepTest [150]) utilize differential and metamorphic testing algorithms for identifying inputs that trigger inconsistencies among different DNN models, or the original and transformed driving scenes. Although such approaches have successfully found various autonomous driving system issues, there still lack approaches that can test DNN-based autonomous driving system with diverse and realistic synthesized driving scenes. Moreover, DeepSafe [60] focuses on automatically identifying safe regions of the input space, within which the network is robust against adversarial perturbations.

3.5.4 GAN-Based Image Translation

GAN-based domain adaption has been recently shown to be effective in unsupervised image-to-image translation [78, 96, 163, 178]. CycleGan [178], DiscoGAN [78] and DualGan [163] propose the similar idea that image-to-image translation should satisfy the cycle consistency, where an image from Domain A should be identical when it is translated to Domain B and translated back to A. The experiments show that this extra constraint can make the translated images more realistic. UNIT [96] further assumes that the representations of two domains may be projected to the same vector space (shared latent space), and is constructed based on VAEs and GANs. Specifically, they

also apply cycle consistency to the GAN model to regularize the translation.

Moreover, GAN-based domain adaption is also applied for virtual-to-real and real-to-virtual driving scene adaption [92, 162]. DU-drive [162] proposes an unsupervised real to virtual domain unification framework for end-to-end driving. Their key insight is the raw image may contain nuisance details which are not related to the prediction of steering angles, and a corresponding virtual scene can ignore these details and also address the domain shift problem. Grad-GAN [92] is designed to automatically transfer the scene annotation in virtual-world to facilitate real-world visual tasks. In that work, a semantic-aware discriminator is proposed for validating the fidelity of rendered image w.r.t each semantic region.

Chapter 4

DeepCheck: Symbolic Execution-Based Attribution and Attack Synthesis for Neural Networks¹

Deep Neural Networks (DNN) are increasingly used in a variety of applications, many of them with substantial safety and security concerns [89]. Our focus in this work is on *image classifiers*: DNNs that take in complex, high dimensional input, pass it through multiple layers of transformations, and finally assign to it a specific output label. Such networks are now being integrated into the perception modules of autonomous or semi-autonomous vehicles, at major car companies such as Tesla, BMW, Ford, and others. It is expected that this trend will continue and intensify, with neural networks being increasingly used in safety critical applications which require high assurance guarantees.

Thus, the traditional emphasis on obtaining high accuracy for DNNs is being augmented with safety and security goals [69]. However, validating DNNs

¹This chapter is based on *Symbolic Execution for Importance analysis and Adversarial generation in Neural Networks* [63]. This work is collaborated with Divya Gopinath, Kaiyuan Wang, Burak Kadron, Corina S. Păsăreanu and Sarfraz Khurshid. The contributions of Mengshi Zhang are discussing and purposing ideas, conducting experiments of symbolic execution and reviewing and revising the paper.

is complex and challenging, due to the nature of the learning techniques that create these models. For example, it is not well understood *why* a DNN, say an image classifier, gives a particular output. This inability to explain the DNN decisions hinders their application in safety critical domains, such as autonomy. Furthermore, evaluating the robustness of a network, e.g., against conceptually simple yet effective attacks, such as 1-pixel attacks, where just one pixel on an image is altered to make the network classify the image incorrectly, is a hard technical problem, due to the huge input space of all images.

This work presents an approach for the analysis of deep neural networks based on *symbolic execution* [40, 79]. Symbolic execution is a well-known program analysis technique that has seen many advances in recent years [32, 33, 56, 76, 136] and applications in various domains, such as security [36, 43], smartphone apps [23], operating systems [161], and databases [47].

Traditional symbolic execution executes programs on *symbolic*, instead of concrete, inputs and systematically explores the program paths (up to a given depth bound). For each path explored, it builds *path conditions*, i.e., constraints on program inputs that execute that path based on the conditional branches in the code. To illustrate, when a conditional statement, say “*if*(*c*)...” is executed, each of the two conditional branches is individually explored, and the path condition *PC* is updated to $PC \wedge c$ for the *then* branch and to $PC \wedge \neg c$ for the *else* branch. The feasibility of the path conditions is checked using off-the-shelf constraint solvers, such as satisfiability modulo theories (SMT) solvers [27, 44], as branch conditions are encountered during

symbolic execution to detect and avoid infeasible paths (if possible) and to generate test inputs that execute feasible paths (as desired). Overall, the program effects are computed as *functions* over the symbolic inputs.

Symbolic execution for deep neural networks is attractive because it would allow us to extract mathematical characterizations (in the form of path conditions and symbolic expressions) of the *internal* behavior of the networks, which are notoriously opaque. The information computed with symbolic execution – even without constraint solving – can be used to examine the *coverage* of the neural network, to extract *explanations* of behavior, and to identify pixels *sensitive to adversarial* behavior. Symbolic execution with constraint solving can also be used to generate *new inputs* to test the networks and find new vulnerabilities.

Our idea is to translate neural networks into an imperative program that can be feasibly analyzed using core ideas from symbolic execution. This translation also has the benefit that other program analysis techniques, in addition to symbolic execution, can be applied to the neural networks. For example, neuron coverage can be analyzed with standard tools that measure branch coverage in software.

To realize this idea, we observe that several popular classes of neural networks use rectified linear units (ReLUs) activation functions and max pooling operations, in the case of convolutional neural networks, that can be naturally translated into a branching structure of the form “*if* ($x > 0$) ... *else* ...”, and thus, a path through the neural network can conceptually be viewed as a

path through the translated program. Hence, path conditions for paths through the network can be built using symbolic execution of the corresponding program, and analyzed as needed.

We apply symbolic execution to select program paths of interest, e.g., paths taken by specific input images, say from the network’s training data. However, just building the path condition for even one path, say using a straightforward application of concolic (or dynamic symbolic) execution [32, 33, 56, 136], can take a considerable amount of time, and solving a path condition with just one symbolic variable can stress modern SMT solvers due to the parsing and simplifications the solver must do to handle constraints, which are conceptually rather simple but syntactically quite complex. We introduce efficient techniques for building path conditions and utilizing solvers. Our approach embodies a lightweight analysis that offers insights into the *reasoning* performed by the networks to make classification decisions with no constraint solving at all, and allows a directed approach for creating adversarial images for network validation with *minimal* constraint solving.

This work makes the following contributions:

- **Idea.** We introduce the idea of using symbolic execution for analyzing neural networks by translating them to imperative code that is practical to analyze, specifically focusing on important pixel identification, and 1-pixel and 2-pixel attack generation.
- **Approach.** We introduce the DeepCheck approach that is based on

translating neural networks with rectified linear units to analyzable imperative code and embodied by two validation techniques; DeepCheck^{Imp}, which applies symbolic execution for identifying important pixels that intuitively provide explanations for why neural networks make certain decision; and DeepCheck^{tPA}, which applies symbolic execution to create 1-pixel and 2-pixel attacks.

- **Evaluation.** We present an experimental evaluation to address three key research questions using the MNIST dataset that consists of images of numeric digits and has been widely studied in literature. The experimental results show that it is feasible to use symbolic execution to identify important pixels and to create 1-pixel and 2-pixel attacks, and that important pixels enable a more scalable approach for generating 1-pixel and 2-pixel attacks. For example, for images of 9 out of 10 digits, a 2-pixel attack is found by checking the 2-pixel combinations of just top-4 important pixels, i.e., a pair of pixels to attack the network is found by checking no more than 6 pixel pairs.

Although 1-pixel and 2-pixel attacks have been studied before [144], we make the surprising new observation that neural networks can be vulnerable to such attacks even along the paths that follow the *same activations* pattern as the original input. Such attacks went unnoticed with previous testing techniques [122, 145, 149], which focused on generating tests that increase the coverage of activated neurons, and hence did not check for attacks along the

same path. Furthermore, if such attacks are shown to not exist, our tool then is able to provide *guarantees* that the network is behaving as expected.

4.1 Background: Neural Networks

Neural networks are often used as *classifiers*, meaning that they assign to each input an output label/class. Such a neural network F can be regarded as a function that assigns to input x an output label y , denoted as $F(x) = y$. Internally, a neural network is comprised of multiple layers of nodes, called neurons. Each node refines and extracts information from values computed by nodes in the previous layer. The first layer is the *input* layer, which takes in the input variables (also called features). The network may have several *hidden* layers: each of its neurons computes a weighted sum of the input variables according to a unique weight vector and a bias value, and then applies an *activation function* to the result ($h_{W,b}(x) = f(\sum_{i=1}^3 W_i x_i + b)$). Most recent networks use rectified linear units (ReLUs) activation functions. A rectified linear unit has output 0 if the input is less than 0, and raw output otherwise; $f(x) = \max(x, 0)$. The last layer uses a *softmax* function to assign an output class to the input. The softmax function squashes the outputs of each node of the previous layer to be between 0 and 1, equivalent to a categorical probability distribution. The number of nodes in this layer is equal to the number of output classes and their respective outputs gives the probability of the input being classified to that class.

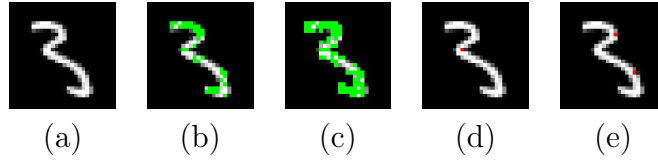


Figure 4.1: (a) Example Image with Predicted Label 3. (b) Top-5% and (c) Top-10% Important Pixels (Green) Identified by DeepCheck^{Imp} ; (d) 1-pixel Attack (Red) Identified by DeepCheck^{tPA} , Changing the Red-pixel to black Changes the Predicted Label to 8; (e) 2-pixel Attack (Red) that Does Include an Attackable Pixel for 1-pixel Attack.

4.2 Overview

This section gives an illustrative overview of our approach to demonstrate our reduction of deep neural nets to imperative code and use of symbolic execution. Our subject neural network $\mathcal{N}$ is a fully connected network, which has been trained on all 60,000 images in the training data of the MNIST dataset [104], and has an accuracy of 92%.

Given the trained network $\mathcal{N}$, we apply our technique DeepCheck^τ to translate it to an imperative program $\mathcal{P}$ that has the same behavior as the original network but is amenable to program analysis. Figure 4.1(a) shows an example image $\mathcal{I}$ from the standard MNIST training data, which has the predicted label of 3 (which is the same as its true label).

4.2.1 Identifying Important Pixels

Given $\mathcal{I}$ as an input, our important pixel identification technique DeepCheck^{Imp} executes the program $\mathcal{P}$, and for the execution path taken by $\mathcal{I}$, computes for every output label, a linear expression in terms of the input variables which

are the 784 pixels of the input image. The algorithm then uses the coefficients of the input pixels in the expression corresponding to the label assigned by the network (3 in the case of the example), to assign an *importance score* for every pixel. A pixel p_1 is considered more important than another p_2 , if the classification decision is impacted more by p_1 than p_2 . DeepCheck^{Imp} employs three metrics; *abs*, *co*, *coi*, to calculate the importance of each pixel. The pixels are then sorted in the descending order of their scores. The pixels which are higher on this list (top threshold %) are identified as being *important*. The insight is that the short-listed important pixels can be held responsible for the classification decision or the classification can be explained or attributed to them. Therefore, a small change to the image with respect to the important pixels, such as changing the value of just one important pixel can have a high impact on the classification decision, and sometimes may lead to the discovery of *adversarial* examples – the new image differs from the original image by the value of just one pixel but this makes the network incorrectly assign a different label to this image.

Figure 4.1(b) illustrates the top-5%, i.e., 39, important pixels highlighted in green. Note, how the important pixels trace the shape of the digit 3 and do not point to areas of the image irrelevant to the digit being identified as 3 such as the background or the edges. Figure 4.1(c) illustrates the top-10%, i.e., 78, important pixels highlighted in green. These important pixels form a denser pattern that traces the shape of the digit 3. This highlights that short-listing pixels based on their importance score in terms of the coefficients of the

expression for the expected label, enables the identification of pixels that can *explain* the classification decision.

4.2.2 Identifying Attack Pixels

Conceptually, our t -pixel attack technique DeepCheck^{tPA} aims to create a new image that differs from the original image at t pixels, and has (1) the same *activation pattern* as the original image but (2) a different label from the original image. Specifically, for a 1-pixel attack, DeepCheck^{tPA} selects a pixel p , makes its value symbolic p_s , retains the original concrete values for all other pixels, and constructs a constraint-solving problem, which requires (1) execution of the same path up to the output layer as the original image $\mathcal{J}$ and (2) change in the output label from the original predicted label of $\mathcal{J}$. The constraint-solving problem consists of a simplified path condition for image $\mathcal{J}$'s execution path such that the path condition contains only one symbolic value, i.e., p_s . If the constraint is satisfiable, a solution provides the value for pixel p to create the 1-pixel attack image. For solving constraints, we use the SMT solver Z3 [44].

DeepCheck^{tPA} checks whether any pixel of $\mathcal{J}$ can be attacked by making one pixel symbolic at a time and checking the resulting path condition. Figure 4.1(d) shows a 1-pixel attack identified by our approach for image $\mathcal{J}$; changing the red pixel to black changes the predicted label of the image to 8. This attackable pixel actually lies in the top-5% (top 39) important pixels for $\mathcal{J}$ identified by DeepCheck^{Imp} . The rank order of this attackable pixel in

descending order of importance is 21. For this case, focusing the 1-pixel attack on important pixels can allow finding an attack much quicker than checking every pixel for attackability. In fact, this image only has one 1-pixel attack. A linear search that starts at the first image pixel (top-left corner) and scans left-to-right takes 346 attempts to find this attack pixel, which is over 16X the attackable pixel’s rank-order (21). We believe *important pixels* can provide a practical heuristic for a *more scalable approach to create attacks*.

To create 2-pixel attacks, we focus DeepCheck^{tPA} on the important pixels identified by DeepCheck^{Imp} , specifically on the top-5% important pixels. We make $\binom{39}{2} = 741$ unordered pairs of the selected important pixels, and for each pair, we make the two corresponding variables symbolic, so each path condition created by symbolic execution contains exactly two symbolic variables. Applying DeepCheck^{tPA} to the 741 pairs results in 93 unique 2-pixel attacks. 38 of the 2-pixel attack pairs contain as an element the pixel that was earlier identified for the 1-pixel attack, whereas 55 of the pairs contain only pixels that are not 1-pixel attackable; Figure 4.1(e) shows one such pair in red.

For this example, the important pixels identified by DeepCheck^{Imp} play a key role in focusing DeepCheck^{tPA} to find a 2-pixel attack. The first attack found by DeepCheck^{tPA} includes the 2 of the 3 top-most important pixels. Thus, the search for a 2-pixel attack for this example requires checking no more than just $\binom{3}{2} = 3$ pairs.

These results illustrate the potential of using symbolic execution in identifying important pixels and creating 1-pixel and 2-pixel attacks, as well

as the value of important pixels in finding attackable pixels and pixel-pairs.

4.3 Technique

We build a symbolic execution framework that i) translates a given neural network into a semantically equivalent imperative program, ii) performs symbolic execution to collect constraints on paths for specific concrete inputs, iii) identifies input features or pixels important to the classification, and iv) synthesizes adversaries that fool the network into misclassification.

4.3.1 Symbolic Execution Framework

The input to our framework is a feed-forward neural network with linear activation functions. Consider network NN with m inputs $X :< x_0, \dots, x_{m-1} >$, n_l layers with l representing each layer, the weights being represented as $w_{i,j}^l$, i.e., the weight of the edge connecting the node j in layer l to node i in layer $l + 1$, b_j^l representing the bias term added to the weighted sum of the input variables from layer l to the node j in layer $l + 1$. Let $Y :< y_0, \dots, y_{n-1} >$ be the output of the nodes before the application of softmax where n is the number of output labels. We consider the second last layer to be the output layer of our model. The output of the classifier is calculated as the index containing the maximum value at the output layer. The goal is to translate NN to an imperative program $\mathcal{P}$ such that for any input x , $NN(x) = \mathcal{P}(x)$. The input state of $\mathcal{P}$ is $S_{inp} =< s_{i_0}, \dots, s_{i_{m-1}} >$, where each s_i corresponds to an input variable x_i , the output state is $S_{op} =< s_{o_0}, \dots, s_{o_{n-1}} >$, where each s_o corresponds to y_i , and the hidden nodes represent the intermediate states $S^h =< s_0^h, \dots, s_{x-1}^h >$ where h is the hidden layer number and v^h is the number

of neurons in layer h .

```

    val := 0 // Initilaization
    for j in range(0, vh-1): // Linear layer
        val := val + wj,ih-1 × sjh-1
    val := val + bi
    if val > 0: // RELU layer
        sih := val
    else:
        sih := 0

```

Figure 4.2: Branching Transformation for Neuron i

Translation. A typical neural network structure does not have any branching. However, observe that in the case of rectified linear units (ReLU), the activation function $f(x) = \max(0, x)$ can be naturally translated into a branching instruction, *if* $(x > 0)$ *then return* x ; *else return* 0 ; . Thus, a path through the neural network can be seen as a path through the translated program, where each executed branch corresponds to the neural network node being activated or not. Every neuron, which is the basic computational unit of the network, applies the function, $h_{W,b}(x)$ on its input x (section 4.1). The code-snippet corresponding to a node i in layer h is shown in Figure 4.2, where v^{h-1} represents the number of nodes at layer $h-1$. This code is invoked v times for every node of the layer h . The corresponding outputs are fed as inputs to the subsequent layer, until the layer before softmax to calculate $S_{op} = \langle s_{o_0}, \dots, s_{o_{n-1}} \rangle$.

Analysis. Once we translate a given NN to a corresponding semantically equivalent $\mathcal{P}$, we perform execution in concolic mode on a given concrete input.

The sections below describe this process and explain how it is used to identify important input attributes and synthesize attacks using constraint solving.

4.3.2 Identifying Important Pixels

This section describes in detail the algorithm of DeepCheck^{Imp} to short-list *important pixels* that could act as *explanations* for the classification decision and also identify pixels *vulnerable to adversaries*.

The function representing a multiple-layer neural network could be expressed in terms of the weights and biases for every layer. Consider the network NN as defined earlier. Let a_j^h represent the intermediate outputs of the hidden nodes in layer h and node j . For instance, for a three layer network, presented below would be the representation of the output in terms of the weights and biases of the network.

$$\begin{aligned} a_0^1 &= f(w_{0,0}^0 \cdot x_0 + w_{0,1}^0 \cdot x_1 + w_{0,2}^0 \cdot x_2 + b_0^0) \\ a_1^1 &= f(w_{1,0}^0 \cdot x_0 + w_{1,1}^0 \cdot x_1 + w_{1,2}^0 \cdot x_2 + b_1^0) \\ a_2^1 &= f(w_{2,0}^0 \cdot x_0 + w_{2,1}^0 \cdot x_1 + w_{2,2}^0 \cdot x_2 + b_2^0) \\ y_0 = a_0^2 &= f(w_{0,0}^1 \cdot a_0^1 + w_{0,1}^1 \cdot a_1^1 + w_{0,2}^1 \cdot a_2^1 + b_0^1) \\ y_1 = a_1^2 &= f(w_{1,0}^1 \cdot a_0^1 + w_{1,1}^1 \cdot a_1^1 + w_{1,2}^1 \cdot a_2^1 + b_1^1) \end{aligned}$$

In the case of a network having linear activations such as $f(x)$ being a ReLU function, the network is a linear model. Each element of Y could be expressed in the form of a linear polynomial expression in terms of the input variables.

For instance, in the previous equation for y_0 , we could remove the function term and replace the values of a_j^h with their respective equations in terms of the input variables (assuming each a_j^h evaluates to a value greater than or equal to 0).

$$\begin{aligned}
y_0 = & w_{0,0}^1 \cdot (w_{0,0}^0 \cdot x_0 + w_{0,1}^0 \cdot x_1 + w_{0,2}^0 \cdot x_2 + b_0^0) \\
& + w_{0,1}^1 \cdot (w_{1,0}^0 \cdot x_0 + w_{1,1}^0 \cdot x_1 + w_{1,2}^0 \cdot x_2 + b_1^0) \\
& + w_{0,2}^1 \cdot (w_{2,0}^0 \cdot x_0 + w_{2,1}^0 \cdot x_1 + w_{2,2}^0 \cdot x_2 + b_2^0) \\
& + b_0^1
\end{aligned}$$

In general in a feed-forward network with ReLU activation functions, each output element, y_i , could be expressed as

$$y_i = C_{i,0} \cdot x_0 + C_{i,1} \cdot x_1 + \dots + C_{i,n-1} \cdot x_{n-1} + b_i$$

, where b_i is the summation of bias terms, and $C_{i,0}$ to $C_{i,n-1}$ are coefficients (signed) of the linear polynomial, that can be calculated in terms of the weights of the non-zero edges from x_j to y_i as shown below;

$$C_{i,j} = \sum_{p \in S_{(i,j)}^{path}} \left(\prod_{e \in S_p^{edge}} w(e) \right) \quad (4.1)$$

where $S_{(i,j)}^{path}$ denotes the set of paths from input node x_j to output node y_i , S_p^{edge} denotes the set of edges on path p , and $w(e)$ denotes the weight of edge e .

DeepCheck^{Imp} Algorithm. Given a concrete input value $\mathcal{I}$ and a translation of the neural network into a program $\mathcal{P}$, DeepCheck^{Imp} instruments the code such that execution of the program on the input also simultaneously updates its impact on the input variables, thereby finally computing the coefficients for the input pixels in the output expression corresponding to the classification decision, y_{label} . It then uses them to identify important pixels. Let us consider the program $\mathcal{P}$ as defined before. Each of these state variables is data-dependent on the input variables. Therefore, we maintain a coefficient array of the size of the input variables corresponding to each of the state-variables, $coef_{i,j}^h$, where h is the layer number (0 to the second-last layer), i is the neuron or node number at that layer (0 to $v^h - 1$) and j is the index of the input variable (0 to $m - 1$).

Given a concrete input $\mathcal{I}$, the coefficient array is initialized as follows $coef_{i,j}^0 = 1$ if $i = j$, 0 otherwise. The execution follows the path taken by the concrete input, i.e. path conditions are evaluated as per the input values. Every time an operation is encountered during the execution of the path, the coefficient arrays are updated accordingly. Consider the code-snippet (Figure 4.3) showing the operations performed to calculate the concrete value of s_i^h and the corresponding update to coefficients $coef_{i,0}^h, \dots, coef_{i,m-1}^h$. Note that since the calculation is done dynamically as the network is being executed on the given input, the coefficient arrays are updated based on the actual values of the activation functions. For instance, wherever the ReLU activation function evaluates to a zero, the respective coefficient arrays get reset to zeros.

```

val := 0 // Initilaization
coeftemp := 0
for j in range(0, vh-1): // Linear layer
    val := val + wj,ih-1 × sjh-1
    coeftemp := coeftemp + wj,ih-1 · coefjh-1
val := val + bi
if val > 0: // RELU layer
    sih := val
    coefih := coeftemp
else:
    sih := 0
    coefih := coeftemp · 0

```

Figure 4.3: Updating the *coef* Arrays. ($\cdot$ is the dot-product of vectors).

After the execution of the first hidden layer 1, for each neuron i the coefficients corresponding to each input variable would be $w_{0,i}^0, w_{1,i}^0, \dots, w_{m-1,i}^0$ respectively. Similarly after the execution of the second hidden layer 2, for each neuron i , the coefficients corresponding to each input variable would be $\sum_{j=0}^{v^1} w_{j,i}^1 \cdot w_{0,i}^0, \dots, \sum_{j=0}^{v^1} w_{j,i}^1 \cdot w_{m-1,i}^0$, where the values of $w_{0,k}^0, \dots, w_{m-1,k}^0$ would be equivalent to zeros respectively if s_k^1 evaluates to zero. Therefore, ultimately at the layer before softmax, the coefficients computed for y_{label} corresponding to the each input variable would be the summation of the products of weights along the non-zero edges from each input variable to y_{label} , equivalent to Eq. 4.1. Note that these coefficients may be different from the one extracted from the entire model with no non-zero edges. The coefficients extracted by DeepCheck^{Imp} precisely corresponds to the non-zero edges of the network for the given input.

Importance Metrics. Many existing techniques use a gradient-based approach 4.5 to determine the impact of each input variable on specific output variables. In a linear model, the partial derivative of an output variable w.r.t an input variable dy_i/dx_j precisely corresponds to the corresponding coefficient $C_{i,j}$. Therefore, we use the value of the coefficient of the input variable to determine its impact on the output variable, akin to gradient-based approaches that use the derivative to determine the impact of each input variable. More specifically, we consider the coefficients of the output variable corresponding to the label assigned by the network to the input, y_{label} . We calculate an important score for each input variable w.r.t. three metrics presented below;

- **co:** We consider the actual values of the coefficients (signed partial derivative dy_{label}/dx_j) to determine the impact of the input variables on y_{label} . This metric would assign higher scores to input variables that would impact y_{label} for all inputs semantically equivalent to the given input; trigger the same activations for the network but may have different values for the input variables. For the example Figure 4.1, the following were the top ten input pixels with the corresponding *co* values (***pixel***, *value*): (**742**, 1.137), (**489**, 1.013), (**488**, 1.006), (**434**, 0.914), (**286**, 0.905), (**458**, 0.887), (**666**, 0.879), (**408**, 0.832), (**377**, 0.825), (**383**, 0.816), (**739**, 0.805).
- **coi:** In order to identify pixels that can be attributed to the classification decision for the specific given input, we need to consider the input

values as well. This can be determined by multiplying the co-efficient values with the corresponding input values, similar to techniques such as DeepLIFT [138] ($x_j \cdot dy_i/dx_j$). These would serve as *better explanations* for the specific input to be assigned the specific label. For the example Figure 4.1, the following were the top ten input pixels with the corresponding *coi* values (***pixel***, *value*): (**434**, 1.365), (**405**, 1.076), (**318**, 0.901), (**292**, 0.890), (**662**, 0.881), (**237**, 0.826), (**433**, 0.788), (**209**, 0.776), (**403**, 0.773), (**463**, 0.695), (**177**, 0.625).

- ***abs***: The absolute values of the coefficients for the expression corresponding to y_{label} represent the magnitude of the impact that a change in the respective input variable would have on the value of y_{label} , irrespective of whether it causes an increase or decrease. Therefore, akin to saliency maps, the absolute value of coefficients could aid in identifying input variables that the classification is most vulnerable to. A small change in the value of these variables can lead to the decision changing thus potentially identifying *adversarial examples*. For the example Figure 4.1, the following were the top ten input pixels with the corresponding *abs* values (***pixel***, *value*): (**742**, 1.136), (**489**, 1.013), (**488**, 1.006), (**415**, 0.986), (**441**, 0.978), (**434**, 0.914), (**286**, 0.905), (**482**, 0.899), (**458**, 0.887), (**666**, 0.879), (**293**, 0.872).

The input variables are ordered in descending order of their importance scores and the pixels corresponding to the top threshold % of the scores are

short-listed as being *important*. The value of the threshold can be user-defined. We have experimented with 5, 10 and 30, respectively.

4.3.3 Synthesizing Attacks

For a given input image $\mathcal{I}$, DeepCheck^{tPA} sets t pixels with symbolic values and the rest pixels with concrete values. Then, it collects path condition of $\mathcal{I}$ for the neural network program $\mathcal{P}$ right before the softmax layer, i.e. layer Y . The final path condition is a conjunction of inequalities (introduced by the ReLU function) of the form $PC = \bigwedge_{h=1}^H (B^h + \sum_{i=1}^t C_i^h \cdot X_i \gamma 0)$, where h represents the h^{th} activation function defined by the computation order, H is the total number of activation functions. In our fully connected network $\mathcal{N}$, each hidden neuron corresponds to an ReLU activation function and we use the same computation order of the activation function to refer hidden neurons. B^h is the bias term of the output value of the h^{th} hidden neuron; C_i^h and X_i are the i^{th} coefficient and symbolic value of the h^{th} hidden neuron, respectively. $\gamma \in \{>, \leq\}$ and is determined by the activeness of the h^{th} hidden neuron. The path condition determines the neuron activation pattern from the network point of view. In practice, the number of conjunct clauses is smaller than H because sometimes all coefficients of the symbolic values (C_i^h) are 0 in which case the entire conjunct clause evaluates to true.

The output value of the j^{th} (0 to n-1) neuron in layer Y is a function of symbolic values of the form $f_j(X) = B^j + \sum_{i=1}^t C_i^j \cdot X_i$. Assume that the network predict label l (0 to n-1) for the input $\mathcal{I}$, then DeepCheck^{tPA} add

constraints $AC = \bigwedge_{j=1, j \neq l'}^n f_j(X) < f_{l'}(X)$ to require the network to predict a label l' where $l \neq l'$. Additionally, DeepCheck^{tPA} add constraints $RA = \bigwedge_{i=1}^t lo \leq X_i \leq hi$, where lo and hi are the lower and upper bound of input values after data normalization. This is to make sure the solution of X_i is in the range. DeepCheck^{tPA} invokes Z3 with constraints $PC \wedge AC \wedge RC$ to solve for concrete values for all X_i . If a solution is found, DeepCheck^{tPA} succeeds in a t pixel attack by setting X with the concrete values Z3 returns and the network predicts label l' which is different from the original predicted label l with the same neuron activation pattern.

4.4 Evaluation

This section describes an experimental evaluation of DeepCheck, specifically the important pixel identification technique DeepCheck^{Imp} and the t -pixel attack technique DeepCheck^{tPA} . We address three key research questions:

- **RQ1.** Does symbolic execution enable important pixel identification? (Section 4.4.2)
- **RQ2.** Does symbolic execution enable 1-pixel attacks and 2-pixel attacks? (Section 4.4.3)
- **RQ3.** How do important pixels identified by DeepCheck^{Imp} compare with pixels that can be attacked using DeepCheck^{tPA} ? (Section 4.4.4)

4.4.1 Subject Neural Network and Images

As our subject neural net, we use the image classification network that is described in Section 4.2. Recall, it is a fully connected $784 \times 10 \times 10 \times 10 \times 10$ network, which has been trained on all 60,000 images in the training data of the MNIST dataset [104], and has an accuracy of 92%. We use 10 images from the training data as subject base images for our evaluation; the images cover all 10 labels $0, 1, \dots, 9$. Figure 4.4 graphically displays them.

4.4.2 Important Pixel Identification

For each image, we apply DeepCheck^{Imp} to compute a ranked list of pixels according to their relative importance based on three ranking metrics: *abs*,



Figure 4.4: Ten Images from the MNIST Training Dataset [104]

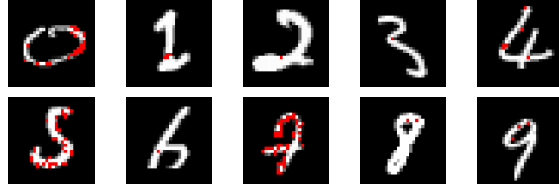


Figure 4.5: Attackable Pixels for 1-pixel Attack Highlighted in Red
co, *coi*. Table 4.1 graphically shows the results produced by DeepCheck^{Imp} for the three metrics for top-5% and top-10% of important pixels. For each image (digit), the table displays the important pixels in green. Table 4.2 displays the results for top-30% of the important pixels with respect to the three metrics.

For top-5% and top-10% results, each metric generally identifies pixels in the central part of the image as most important, with top-10% forming a denser pattern than top-5%. The *abs* and *co* metrics show similar patterns in the central region. The *coi* metric most closely follows the digit’s shape.

Overall, the use of symbolic execution enables identification of important pixels that help explain the classification decisions of the neural network.

4.4.3 *t*-pixel Attack

We apply DeepCheck^{tPA} for 1-pixel attack and 2-pixel attack. For 1-pixel attack, for each image, we evaluate each of the 784 pixels in the image

digit	5%(39)			10%(78)		
	abs	co	coi	abs	co	coi
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						

Table 4.1: Top-5% and Top-10% of Important Pixels (Green) Identified by DeepCheck^{Imp} for *abs*, *co*, and *coi*.

to determine if it is *attackable*, i.e., can be given a different value to change the image’s predicted label while preserving the neuron activation pattern. Figure 4.5 highlights, for each digit, each attackable pixel (in red) for a 1-pixel attack. The attackable pixels lie on or very close to the shape of the

digit	abs	co	coi
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			

Table 4.2: Top-30% (i.e., 235) of Important Pixels (Green) Identified by DeepCheck^{Imp} for *abs*, *co*, and *coi*.

corresponding digit.

Table 4.3 summarizes the attacks identified by DeepCheck^{tPA} for 1-pixel attack. Some images, e.g., digit 2, contain one attackable pixel out of 784 pixels, whereas some others contain multiple, e.g., 47 for digit 7. All images except digit 6 when attacked get a unique incorrect label. Digit 6 has 2 attacks

digit	0	1	2	3	4	5	6	7	8	9
#ap	25	4	1	1	6	36	1	47	2	3
alabel	5	3	3	8	6	3	5,8	9	1	4
1stap	244	489	516	346	71	103	486	156	211	240

Table 4.3: Number of 1-pixel Attackable Pixels (#ap) Identified by DeepCheck^{tPA} for Each Image (digit) among All Pixels. “alabel” shows the set of labels that the network incorrectly produces after the 1-pixel attacks. “1stap” shows the smallest position (pixel ID) of any attackable pixel in the image. For digit 6, the pixel at position 486 can be attacked to become 5 as well as 8 by appropriately setting the pixel’s value.

digit	0	1	2	3	4	5	6	7	8	9
#a2p	548	198	48	93	260	463	287	651	111	171
#a2p-new	60	87	10	55	114	100	186	75	36	96
alabel	5	2,3	3	8	6	3	1,3,5,8	8,9	1,2,3	4

Table 4.4: Number of 2-pixel Attacks (#a2p) Identified by DeepCheck^{tPA} for Each Image (digit), and of These Attacks the Number that do not Use Any Pixel that is Attackable for a 1-pixel Attack (#a2p-new). “alabel” shows the set of labels that each digit can be attacked to incorrectly produce using the neural net. The search for 2-pixel attacks is focused on top-5% important pixels identified by the *coi* metric of DeepCheck^{Imp}.

but both use the same pixel, which can be attacked in two ways: to incorrect label 5 and incorrect label 8. The row “1stap” shows the smallest position of any attackable pixel for each digit; this position represents the number of attempts DeepCheck^{tPA} takes to find the first attackable pixel for each image when it exhaustively checks every pixel in the image. For some digits, e.g., 4, less than 10% of the pixels are checked. For some other digits, e.g., 1, over 62% of the pixels are checked. Note, this exhaustive search is without utilizing the important pixel identification, which we utilize next.

digit	#ap	5%(39)			10%(78)			30%(235)		
		abs	co	coi	abs	co	coi	abs	co	coi
0	25	12.0	28.0	64.0	28.0	44.0	68.0	80.0	64.0	68.0
1	4	0.0	25.0	75.0	25.0	50.0	100.0	75.0	100.0	100.0
2	1	0.0	0.0	100.0	0.0	100.0	100.0	100.0	100.0	100.0
3	1	0.0	0.0	100.0	0.0	0.0	100.0	0.0	100.0	100.0
4	6	66.7	50.0	66.7	66.7	50.0	66.7	100.0	66.7	66.7
5	36	2.8	5.6	30.6	11.1	8.3	50.0	38.9	33.3	52.8
6	1	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
7	47	19.1	17.0	40.4	29.8	25.5	48.9	48.9	40.4	48.9
8	2	0.0	0.0	100.0	0.0	0.0	100.0	100.0	100.0	100.0
9	3	0.0	0.0	66.7	0.0	0.0	66.7	66.7	66.7	66.7

Table 4.5: Important Pixels and Attackable Pixels

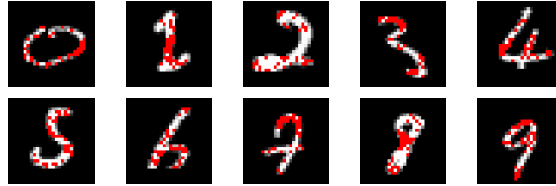


Figure 4.6: Attackable Pixels for 2-pixel Attack Highlighted in Red. For each digit, the union of all pixels that are part of any 2-pixel attack is shown.

For 2-pixel attack, for each image, we select the top-5% important pixels identified by the *coi* metric in DeepCheck^{Imp} and select all $\binom{39}{2} = 741$ unordered pairs that can be formed using the important pixels selected, and evaluate each pair to determine if it is attackable. Table 4.4 shows the number of attacks identified by DeepCheck^{tPA} 2-pixel attack, and of those attacks the number that does not include any attackable pixel for 1-pixel attack. As expected, many 2-pixel attacks consist of a pixel that was 1-pixel attackable. However, several new attack pairs which do not include any pixel that is attackable for 1-pixel attack are found. 4 out of 10 digits can be attacked to

create multiple incorrect labels (alabel), e.g., digit 8 can be attacked using 3 different 2-pixel attacks to make the neural net incorrectly classify it as 1, 2, or 3.

For each digit, Figure 4.6 shows the union of all pixels in any 2-pixel attack to display their location. These pixels lie on or very close to the shape of the corresponding digit.

Overall, the use of symbolic execution enables finding both 1-pixel and 2-pixel attacks for each image.

4.4.4 Important Pixels and Attackable Pixels

Table 4.5 presents a comparison of important pixels and attackable pixels for 1-pixel attack. For each image (digit), the column “#ap” shows the number of attackable pixels identified by DeepCheck^{tPA} for 1-pixel attack. The group of 3 columns labeled 5% shows the percentage of attackable pixels that are in the top-5%, i.e., 39, of the important pixels identified by each of the 3 metrics of DeepCheck^{Imp}. The next 2 groups of 3 columns show the corresponding results for top-10% and top-30% of important pixels. At the top-5% and top-10% levels, *coi* outperforms the other two metrics in predicting attackability. For 3 of the 10 images, all attackable pixels lie in the top-5% of important pixels identified by *coi*. Moreover, for 6 of the 10 images, at least two-thirds of the attackable pixels are in top-5% for *coi*. Furthermore, for 8 out of 10 images, at least one-half of attackable pixels are in the top-5% for *coi*. At the top-30% level, the *abs* metric and *coi* metric perform similarly and have

digit	abs	co	coi
0	7(0.9%)	5(0.6%)	1(0.1%)
1	60(7.7%)	23(2.9%)	6(0.8%)
2	119(15.2%)	66(8.4%)	19(2.4%)
3	254(32.4%)	169(21.6%)	21(2.7%)
4	3(0.4%)	1(0.1%)	1(0.1%)
5	6(0.8%)	1(0.1%)	1(0.1%)
6	2(0.3%)	2(0.3%)	2(0.3%)
7	4(0.5%)	2(0.3%)	1(0.1%)
8	142(18.1%)	86(11.0%)	19(2.4%)
9	169(21.6%)	98(12.5%)	13(1.7%)

Table 4.6: Smallest Number of Important Pixels to Explore for the First 1-pixel Attack for Each Image (Digit) and Each Metric

digit	0	1	2	3	4	5	6	7	8	9
coi	2	2	19	3	2	2	2	2	4	2

Table 4.7: Smallest Number of Important Pixels to Explore for the First 2-pixel Attack for Each Image for the *coi* Metric

higher effectiveness than *co*, although the difference in the metrics reduces as % of important pixels increases. Overall, the results show that at the 5% and 10% levels, important pixels identified by *coi* likely contain the majority of attackable pixels.

Table 4.6 shows for each image (digit) the smallest number of important pixels that must be explored before a 1-pixel attack is found for each of the three metrics. For all metrics and all images, no more than top one-third of the important pixels need to be checked to find an attack pixel. Moreover, for all metrics, less than 10 pixels need to be checked to discover an attack pixel for at least half of the images. The *coi* metric requires a maximum of 21 pixels to be checked across all the images, and for 4 images, the top most important pixel identified by *coi* is an attack pixel.

Table 4.4 already shows that forming pairs using just top-5% of the important pixels based on the *coi* metric allows several 2-pixel attacks on each image (digit). Table 4.7 presents the smallest number of important pixels to explore to find the first 2-pixel attack for each image (digit) based on the *coi* metric. The worst case is for digit 2, where top-19 important pixels must be considered to find a 2-pixel attack. The best case happens for 7 out of 10 digits, where the top-2 important pixels allow DeepCheck^{*tPA*} to create a 2-pixel attack; for these 7 cases, using DeepCheck^{*Imp*} with *coi* metric presents an optimal strategy to find a 2-pixel attack. Moreover, for 9 out of 10 digits, a 2-pixel attack is found within just the top-4 important pixels, i.e., exploration of no more than 6 pairs.

Overall, important pixel identification using DeepCheck^{*Imp*} plays an important role in focusing on the exploration of DeepCheck^{*tPA*} to find 1-pixel and 2-pixel attacks.

4.5 Related Work

Recent independent work, developed concurrently with ours, proposes concolic testing for deep neural networks [145]. However, their focus is on defining and achieving test coverage requirements, although their approach also produces adversarial images. In contrast we use symbolic execution for identifying important pixels and for specific 1-pixel and 2-pixel attacks, which target the same activation pattern as the original image; furthermore, we use important pixels to focus the search for attackable pixels.

Other related recent techniques include formal methods [69, 75] and testing [122, 149] for deep neural networks. However none of the previous work uses formal methods for important pixel identification, or more generally for explainability in neural networks. Furthermore, they do not check for attacks along with the same activation patterns.

In our work, we use Z3 as the off-the-shelf constraint solver, which is inherited from a software analysis tool [119]. We note that other constraint solvers can be plugged in our analysis. For example, a good option is Reluplex [69, 75], which has been optimized specifically for the analysis of neural networks with ReLU activations. However, our methodology is general and can be in principle applied to other networks with linear units, such as Convolutional Neural Networks, which can not be handled by Reluplex. Another option is to use linear programming as in [145], since for a fixed activation pattern, the problem to be solved becomes linear. However, linear solvers may behave unexpectedly when no solution exists and can give unsound results due

to overflow [151].

The rest of this section describes existing techniques related to attribution or explainability in neural networks and also existing techniques for adversarial example generation.

4.5.1 Techniques for Attribution

Despite the wide-spread adoption of neural networks, most deep neural network classifiers are black-boxes. It is crucial to understand the reasons behind the predictions of these classifiers in order to build trust in the model. Therefore, a number of techniques have been explored in the area of generating explanations for predictions. *Attribution* is a specific class of approaches, mostly applicable to image classification applications, where the technique attempts to assign "relevance", "contribution" to each input feature or pixel towards the classification decision. We describe below broad categories of attribution approaches.

Perturbation-based approaches alter the value of every input feature individually by a specific amount [166], re-run the network on the input and then measure the difference in the output value. However, these techniques tend to be slow and the computation time increases with the number of features. *Gradient-based approaches* [138]) compute the attributions of every feature in a single forward and backward pass of the network on a given input. They compute the signed partial derivative of the output w.r.t each input variable and multiple it by the input value to determine the impact of that variable

on the output. *Integrated-gradients* [146]) proposed an approach that takes an average of the attributions to calculate along a linear path from a baseline (user-defined) until the given input. *Saliency maps* [139] consider the absolute value of the partial derivatives of the output w.r.t each input variable in order to identify pixels that can perturb the least to observe a sizable change in the output value.

4.5.2 Techniques for Adversarial Attack Generation

It has been observed that state-of-the-art networks are highly vulnerable to *adversarial perturbations*: given a correctly-classified input x , it is possible to find a new input x' that is very similar to x but is assigned a different label [147]. Goodfellow et al. [59] introduced the Fast Gradient Sign Method for crafting adversarial perturbations using the derivative of the model’s loss function with respect to the input feature vector. They show that NNs trained for the MNIST and CIFAR-10 classification tasks can be fooled with a high success rate. An extension of this approach applies the technique in an iterative manner [50]. Jacobian-based Saliency Map Attack (JSMA) [117] proposed a method for targeted misclassification by exploiting the forward derivative of a NN to find an adversarial perturbation that will force the model to misclassify into a specific target class. Carlini et. al. [35] recently proposed an approach that could not be resisted by state-of-the-art networks such as those using defensive distillation. Their optimization algorithm uses better loss functions and parameters (empirically determined) and uses three different distance metrics.

The DeepFool [106] technique simplifies the domain by considering the network to be completely linear. They compute adversarial inputs on the tangent plane (orthogonal projection) of a point on the classifier function. They then introduce non-linearity to the model, and repeat this process until a true adversarial example is found. Deep Learning Verification (DLV) [69] is an approach that defines a region of safety around a known input and applies SMT solving for checking robustness. They consider the input space to be discretized and alter the input using manipulations until it is at a minimal distance from the original, to generate possibly-adversarial inputs. DeepSafe [61] is an approach that first applies a label-guided clustering algorithm on inputs with known labels to identify input regions that can be expected to be consistently labeled. It then employs the Reluplex solver [75] to verify that the all possible inputs within a given region are assigned the same label by the network. DeepRoad [172] introduces an unsupervised learning technique based on DNNs themselves for validating DNN-based autonomous drivers.

Chapter 5

Conclusion

Software testing and machine learning are two active research domains which have co-existed for several decades but evolved largely independently. This thesis is that a novel integration of the approaches which aims at making software testing and machine learning more effective. Specifically, our insight of the integration is bi-directional: in one direction, we can utilize machine learning techniques to enhance finding and removing bugs in various software systems, in the other direction, we can leverage software testing techniques to improve machine learning systems. Based on this insight, we proposed three techniques to explore the opportunities for integrating these two techniques. First, we proposed ePRFL, which uses the PageRank algorithm to boost the accuracy of traditional techniques for localizing bugs in code. Second, we proposed DeepRoad, a GAN-Based Metamorphic Testing and Input Validation framework for DNN-based autonomous driving systems. Finally, we defined an analysis approach based on symbolic execution to study the properties of deep neural networks. We believe our work in this thesis can (1) help developers make fewer efforts on debugging and (2) explore more defects of machine-learning-based software systems to make them more robust.

Bibliography

- [1] Open web application security project: Data validation. https://www.owasp.org/index.php/Data_Validation, 2013. Accessed: Jun. 2018.
- [2] Tesla autopilot system. <https://www.tesla.com/autopilot>, 2014. Accessed: Jun. 2018.
- [3] Steering angle model: Autumn. <https://github.com/udacity/self-driving-car/tree/master/steering-models/evaluation>, 2016. Accessed: Jun. 2018.
- [4] Steering angle model: Chauffeur. <https://github.com/udacity/self-driving-car/tree/master/steering-models/community-models/chauffeur>, 2016. Accessed: Jun. 2018.
- [5] Steering angle model: Rwrightman. <https://github.com/udacity/self-driving-car/tree/master/steering-models/evaluation>, 2016. Accessed: Jun. 2018.
- [6] Udacity pre-trained models. <https://github.com/udacity/self-driving-car/tree/master/steering-models/evaluation>, 2016. Accessed: Jun. 2018.
- [7] Udacity self driving car. <https://github.com/udacity/self-driving-car>, 2016. Accessed: Jun. 2018.
- [8] Asm java bytecode manipulation and analysis framework, 2018. Accessed: 02-10-2018.

- [9] Github a web-based version control repository and internet hosting service, 2018. Accessed: 02-10-2018.
- [10] Java programming language agents, 2018. Accessed: 02-10-2018.
- [11] Maven central repository, 2018. Accessed: 02-10-2018.
- [12] Maven failsafe plugin, 2018. Accessed: 02-10-2018.
- [13] Maven surefire plugin, 2018. Accessed: 02-10-2018.
- [14] Numpy package for scientific computing with python, 2018. Accessed: 02-10-2018.
- [15] Pit mutation testing tool, 2018. Accessed: 02-10-2018.
- [16] Tesla model s crash. <https://www.wired.com/story/tesla-autopilot-why-crash-radar> 2018. Accessed: Jun. 2018.
- [17] Uber’s self-driving cars were struggling before arizona crash. <https://www.nytimes.com/2018/03/23/technology/uber-self-driving-cars-arizona.html>, 2018. Accessed: Jun. 2018.
- [18] Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION*, pages 89–98. IEEE, 2007.
- [19] Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM*

- International Conference on Automated Software Engineering*, pages 88–99. IEEE Computer Society, 2009.
- [20] Mohammad Amin Alipour and Alex Groce. Extended program invariants: applications in testing and fault localization. In *WODA*, pages 7–11. ACM, 2012.
 - [21] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d’Amorim. Fault-localization using dynamic slicing and change impact analysis. In *ASE*, pages 520–523. IEEE, 2011.
 - [22] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.
 - [23] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *SIGSOFT FSE*, page 59. ACM, 2012.
 - [24] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE*, pages 402–411. ACM, 2005.
 - [25] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *ISSTA*, pages 177–188. ACM, 2016.
 - [26] Shane Barratt and Rishi Sharma. A note on the inception score. *arXiv preprint arXiv:1801.01973*, 2018.

- [27] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, July 2011.
- [28] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. Graph-based analysis and prediction for software evolution. In *ICSE*, pages 419–429, 2012.
- [29] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [30] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Praseen Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.
- [31] Allan Borodin, Gareth O Roberts, Jeffrey S Rosenthal, and Panayiotis Tsaparas. Link analysis ranking: algorithms, theory, and experiments. *ACM Transactions on Internet Technology (TOIT)*, 5(1):231–297, 2005.
- [32] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
- [33] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, 2005.

- [34] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d’Amorim. Entropy-based test generation for improved fault localization. In *ASE*, pages 257–267. IEEE, 2013.
- [35] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *IEEE S&P*, 2017.
- [36] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on binary code. In *IEEE S&P*, pages 380–394, 2012.
- [37] Thierry Titchou Chekam, Mike Papadakis, and Yves Le Traon. Assessing and comparing mutation-based fault localization techniques. *arXiv preprint arXiv:1607.05512*, 2016.
- [38] Tsong Y Chen, Shing C Cheung, and Siu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.
- [39] Alexei D Chepelianskii. Towards physical laws for software architecture. *arXiv preprint arXiv:1003.5455*, 2010.
- [40] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE TSE*, 2(3), 1976.
- [41] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with ample. In *AADeBUG*, pages 99–104. ACM, 2005.

- [42] Tung Dao, Lingming Zhang, and Na Meng. How does execution information help with information-retrieval based bug localization? In *ICPC*, pages 241–250. IEEE, 2017.
- [43] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, 2013.
- [44] Leonardo de Moura and Nikolaj Bjorner. Z3: An efficient SMT solver. In *TACAS*, 2008.
- [45] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101. Springer, 1995.
- [46] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *CSTVA*, pages 30–39. ACM, 2014.
- [47] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *ISSTA*, 2007.
- [48] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

- [49] Ayman Farahat, Thomas LoFaro, Joel C Miller, Gregory Rae, and Lesley A Ward. Authority rankings from hits, pagerank, and salsa: Existence, uniqueness, and effect of initialization. *SIAM Journal on Scientific Computing*, 27(4):1181–1201, 2006.
- [50] Reuben Feinman, Ryan R. Curtin, Saurabh Shintre, and Andrew B. Gardner. Adversarial machine learning at scale, 2016. Technical Report. <http://arxiv.org/abs/1611.01236>.
- [51] Min Feng and Rajiv Gupta. Learning universal probabilistic models for fault localization. In *PASTE*, pages 81–88. ACM, 2010.
- [52] Leon Gatys, Alexander S Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 262–270, 2015.
- [53] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2016 IEEE Conference on*, pages 2414–2423. IEEE, 2016.
- [54] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 19–30. ACM, 2019.

- [55] David F Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.
- [56] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, 2005.
- [57] Alberto Gonzalez-Sanchez, Rui Abreu, Hans-Gerhard Gross, and Arjan JC van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In *Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering*, pages 83–92. IEEE Computer Society, 2011.
- [58] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.
- [59] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2014. Technical Report. <http://arxiv.org/abs/1412.6572>.
- [60] Divya Gopinath, Guy Katz, Corina S Pasareanu, and Clark Barrett. Deepsafe: A data-driven approach for checking adversarial robustness in neural networks. *arXiv preprint arXiv:1710.00486*, 2017.
- [61] Divya Gopinath, Guy Katz, Corina S. Pasareanu, and Clark Barrett. DeepSafe: A data-driven approach for checking adversarial robustness in

neural networks, 2017. <https://arxiv.org/abs/1710.00486>.

- [62] Divya Gopinath, Corina S Păsăreanu, Kaiyuan Wang, Mengshi Zhang, and Sarfraz Khurshid. Symbolic execution for attribution and attack synthesis in neural networks. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, pages 282–283. IEEE Press, 2019.
- [63] Divya Gopinath, Mengshi Zhang, Kaiyuan Wang, Burak Kadron, Corina S. Păsăreanu, and Sarfraz Khurshid. Symbolic execution for importance analysis and adversarial generation in neural networks. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019. To appear.
- [64] David Grove and Craig Chambers. A framework for call graph construction algorithms. *TOPLAS*, 23(6):685–746, 2001.
- [65] Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *ASE*, pages 263–272. ACM, 2005.
- [66] Birgit Hofer and Franz Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *ECAI*, pages 420–425, 2012.
- [67] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Mutation-based fault localization for

- real-world multilingual programs (t). In *ASE*, pages 464–475. IEEE, 2015.
- [68] Henry Hsu and Peter A Lachenbruch. Paired t test. *Wiley encyclopedia of clinical trials*, pages 1–3, 2007.
- [69] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *CAV*, 2017.
- [70] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Fault localization using value replacement. In *ISSTA*, pages 167–178. ACM, 2008.
- [71] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European Conference on Computer Vision*, pages 694–711. Springer, 2016.
- [72] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477. ACM, 2002.
- [73] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, pages 437–440. ACM, 2014.
- [74] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE*, pages 654–665. ACM, 2014.

- [75] G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *CAV*, 2017.
- [76] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.
- [77] Myunghwan Kim, Roshan Sumbaly, and Sam Shah. Root cause detection in a service-oriented architecture. In *PER*, volume 41, pages 93–104. ACM, 2013.
- [78] Taeksoo Kim, Moonsu Cha, Hyunsoo Kim, Jungkwon Lee, and Jiwon Kim. Learning to discover cross-domain relations with generative adversarial networks. *arXiv preprint arXiv:1703.05192*, 2017.
- [79] James C. King. Symbolic execution and program testing. *CACM*, 19(7), 1976.
- [80] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.
- [81] Jon M Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632, 1999.
- [82] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners’ expectations on automated fault localization. In *ISSTA*, pages 165–176, 2016.

- [83] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [84] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. Fine-tuning spectrum based fault localisation with frequent method item sets. In *ASE*, pages 274–285. ACM, 2016.
- [85] Tien-Duy B Le, David Lo, and Ming Li. Constrained feature selection for localizing faults. In *ICSME*, pages 501–505. IEEE, 2015.
- [86] Tien-Duy B Le, Richard J Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: better together. In *FSE*, pages 579–590. ACM, 2015.
- [87] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *TSE*, 38(1):54–72, 2012.
- [88] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.
- [89] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nature*, 521(7553), 2015.
- [90] Ronny Lempel and Shlomo Moran. The stochastic approach for link-structure analysis (salsa) and the tlc effect1. *Computer Networks*, 33(1-6):387–401, 2000.

- [91] Nuo Li, Tao Xie, Maozhong Jin, and Chao Liu. Perturbation-based user-input-validation testing of web applications. *Journal of Systems and Software*, 83(11):2263–2274, 2010.
- [92] Peilun Li, Xiaodan Liang, Daoyuan Jia, and Eric P Xing. Semantic-aware grad-gan for virtual-to-real urban scene adaption. *arXiv preprint arXiv:1801.01726*, 2018.
- [93] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–180. ACM, 2019.
- [94] Xia Li and Lingming Zhang. Transforming programs and tests in tandem for fault localization. In *OOPSLA*, pages 92–122. ACM, 2017.
- [95] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15–26, 2005.
- [96] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. Unsupervised image-to-image translation networks. In *Advances in Neural Information Processing Systems*, pages 700–708, 2017.
- [97] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *FSE*, pages 166–178. ACM, 2015.

- [98] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.
- [99] Alexis C Madrigal. Inside waymo’s secret world for training self-driving cars. *The Atlantic*, 2017.
- [100] Xiaoguang Mao, Yan Lei, Ziyang Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.
- [101] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, pages 1–29, 2016.
- [102] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.
- [103] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Guided mutation testing for javascript web applications. *TSE*, 41(5):429–444, 2015.
- [104] The MNIST database of handwritten digits Home Page. <http://yann.lecun.com/exdb/mnist/>.

- [105] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, pages 153–162. IEEE, 2014.
- [106] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. DeepFool: A simple and accurate method to fool deep neural networks. In *CVPR*, 2016.
- [107] Christian Murphy, Gail E Kaiser, Lifeng Hu, and Leon Wu. Properties of machine learning applications for use in metamorphic testing. In *SEKE*, volume 8, pages 867–872, 2008.
- [108] Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. An empirical study of static call graph extractors. *TOSEM*, 7(2):158–191, 1998.
- [109] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *TOSEM*, 20(3):11, 2011.
- [110] Aniruddh Nath and Pedro Domingos. Learning tractable probabilistic models for fault localization. In *AAAI*, pages 1294–1301. AAAI Press, 2016.
- [111] Froilan S Ocariza Jr, Karthik Pattabiraman, and Ali Mesbah. Autofloxx: An automatic fault localizer for client-side javascript. In *ICST*, pages 31–40. IEEE, 2012.

- [112] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.
- [113] Mike Papadakis, Marcio E Delamaro, and Yves Le Traon. Proteum/fl: A tool for localizing faults using mutation analysis. In *SCAM*, pages 94–99. IEEE, 2013.
- [114] Mike Papadakis and Yves Le Traon. Using mutants to locate "unknown" faults. In *ICST*, pages 691–700. IEEE, 2012.
- [115] Mike Papadakis and Yves Le Traon. Effective fault localization via mutation analysis: A selective mutation approach. In *SAC*, pages 1293–1300. ACM, 2014.
- [116] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *STVR*, 25(5-7):605–628, 2015.
- [117] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *EuroS&P*, 2016.
- [118] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209. ACM, 2011.
- [119] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehltitz, and Neha Rungta. Symbolic pathfinder: integrat-

- ing symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.
- [120] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *ICSE*, pages 609–620. IEEE, 2017.
 - [121] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.
 - [122] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *SOSP*, 2017.
 - [123] Alexandre Perez and Rui Abreu. A qualitative reasoning approach to spectrum-based fault localization. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, pages 372–373. ACM, 2018.
 - [124] Alexandre Perez, Rui Abreu, and Marcelo d’Amorim. Prevalence of single-fault fixes and its impact on fault localization. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 12–22. IEEE, 2017.
 - [125] Alexandre Perez, Rui Abreu, and INESC-TEC HASLab. Leveraging qualitative reasoning to improve sfl. In *IJCAI*, pages 1935–1941, 2018.

- [126] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering*, pages 654–664. IEEE Press, 2017.
- [127] Dean A. Pomerleau. Advances in neural information processing systems 1. chapter ALVINN: An Autonomous Land Vehicle in a Neural Network, pages 305–313. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1989.
- [128] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziyang Dai, and Chengsong Wang. The strength of random search on automated program repair. In *ICSE*, pages 254–265. ACM, 2014.
- [129] Bernard Rosner. A generalization of the paired t-test. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 31(1):9–13, 1982.
- [130] Shounak Roychowdhury and Sarfraz Khurshid. Software fault localization using feature selection. In *MALETS*, pages 11–18. ACM, 2011.
- [131] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 10–13. IEEE, 2018.
- [132] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic

- modeling. In *Fifteenth annual conference of the international speech communication association*, 2014.
- [133] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2234–2242, 2016.
 - [134] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortes. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, Sept 2016.
 - [135] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.
 - [136] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, 2005.
 - [137] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *ASE*, pages 201–211. IEEE, 2015.
 - [138] Avanti Shrikumar, Peyton Greenside, Anna Shcherbina, and Anshul Kundaje. Not just a black box: Learning important features through propagating activation differences. *CoRR*, 2016.

- [139] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, 2013.
- [140] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [141] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for java runtime exceptions. In *ISSTA*, pages 153–164. ACM, 2009.
- [142] Jeongju Sohn and Shin Yoo. Fluccs: using code and change metrics to improve fault localization. In *ISSTA*, pages 273–283. ACM, 2017.
- [143] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*, pages 314–324. ACM, 2013.
- [144] Jiawei Su, Danilo Vasconcellos Vargas, and Sakurai Kouichi. One pixel attack for fooling deep neural networks. *CoRR*, abs/1710.08864, 2017.
- [145] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. *arXiv preprint arXiv:1805.00089*, 2018.
- [146] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *ICML*, 2017.

- [147] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks, 2013. Technical Report. <http://arxiv.org/abs/1312.6199>.
- [148] Gregory Tasse. The economic impacts of inadequate infrastructure for software testing. Collingdale, PA, USA, 2002. Diane.
- [149] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. *CoRR*, abs/1708.08559, 2017.
- [150] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, May 27 - June 3, 2018*, ICSE 2018, 2018.
- [151] Yakir Vizel, Alexander Nadel, and Sharad Malik. Solving linear arithmetic with sat-based model checking. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 47–54, 2017.
- [152] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.
- [153] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *TR*, 63(1):290–308, 2014.

- [154] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *TSE*, 42(8):707–740, 2016.
- [155] Xiaoyuan Xie, Joshua Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Application of metamorphic testing to supervised classifiers. In *Quality Software, 2009. QSIC'09. 9th International Conference on*, pages 135–144. IEEE, 2009.
- [156] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.
- [157] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. Provably optimal and human-competitive results in sbse for spectrum based fault localization. In *SSBSE*, pages 224–238. Springer, 2013.
- [158] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *TSE*, 43(1):34–55, 2017.
- [159] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *ICSME*, pages 191–200. IEEE, 2014.

- [160] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *FSE*, pages 52–63. ACM, 2014.
- [161] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *IEEE S&P*, 2006.
- [162] Luona Yang, Xiaodan Liang, and Eric Xing. Unsupervised real-to-virtual domain unification for end-to-end highway driving. *arXiv preprint arXiv:1801.03458*, 2018.
- [163] Zili Yi, Hao Zhang, Ping Tan, and Minglun Gong. Dualgan: Unsupervised dual learning for image-to-image translation. *arXiv preprint*, 2017.
- [164] Shin Yoo. Evolving human competitive spectra-based fault localization techniques. In *SSBSE*, pages 244–258. Springer, 2012.
- [165] Kai Yu, Mengxiang Lin, Qing Gao, Hui Zhang, and Xiangyu Zhang. Locating faults using multiple spectra-specific models. In *SAC*, pages 1404–1410. ACM, 2011.
- [166] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *ECCV*, 2014.
- [167] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 23–32. IEEE, 2011.

- [168] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, pages 765–784, 2013.
- [169] Mengshi Zhang. Boosting spectrum-based fault localization using pagerank. Master’s thesis, 2017.
- [170] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *ISSTA*, pages 261–272. ACM, 2017.
- [171] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. An empirical study of boosting spectrum-based fault localization via pagerank. *IEEE Transactions on Software Engineering*, 2019.
- [172] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.
- [173] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *ICSE*, pages 272–281. ACM, 2006.
- [174] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faulty code

- by multiple points slicing. *Software: Practice and Experience*, 37(9):935–961, 2007.
- [175] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. A study of effectiveness of dynamic slicing in locating real faults. *Empirical Software Engineering*, 12(2):143–160, 2007.
- [176] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *AADeBUG*, pages 33–42. ACM, 2005.
- [177] Zhi Quan Zhou, DH Huang, TH Tse, Zongyuan Yang, Haitao Huang, and TY Chen. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*, pages 346–351, 2004.
- [178] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv preprint arXiv:1703.10593*, 2017.