HIGH-QUALITY AUTOMATIC PROGRAM REPAIR

A Dissertation Presented

by

MANISH MOTWANI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2022

Robert and Donna Manning College of
Information and Computer Sciences

© Copyright by Manish Motwani 2022

All Rights Reserved

HIGH-QUALITY AUTOMATIC PROGRAM REPAIR

A Dissertation Presented

by

MANISH MOTWANI

Approved as to style and content by:

Yuriy Brun, Chair

Arjun Guha, Member

George S. Avrunin, Member

Claire Le Goues, Member

James Allan, Chair of the Faculty
Robert and Donna Manning College of
Information and Computer Sciences

DEDICATION

I dedicate this work to my divine Satguru (True Guru) for providing me the perseverance to solve real-life challenges and make me grow intellectually and personally. And to my parents, Harish and Neeta Motwani, for their unconditional love and support.

ACKNOWLEDGMENTS

Words cannot express my gratitude to my advisor, Prof. Yuriy Brun, for providing me an opportunity to work with him, connecting me with other great minds in the field, for his invaluable patience, support, and feedback in finishing my doctorate, and for inspiring and helping me to pursue a career in academia. During my PhD journey, he taught me many things, including identifying and solving good research problems, effectively communicating my thoughts and ideas, amicably working with others, and most importantly, being a good, honest, and empathetic person. He has not only been a great advisor but also a great friend, supporting me through the difficult times in my personal and professional life. I look forward to continuing his legacy for my future students. Thank you! Yuriy.

I could not have undertaken this journey without my dissertation committee members, Prof. Claire Le Goues, Prof. Arjun Guha, and Prof. George S. Avrunin, who generously provided their knowledge, feedback, and expertise. I especially thank Prof. Claire Le Goues, whose foundational and pioneering work on automatic program repair is the inspiration for my doctoral research. I was fortunate to collaborate with her during my PhD, and I look forward to working together again in the future. Additionally, this endeavor would not have been possible without the generous support from the National Science Foundation and the University of Massachusetts Amherst, especially during the pandemic.

I extend my thanks to my research collaborators, Sandhya Sankaranarayanan, Afsoon Afzal, Mauricio Soto, Prof. Kathryn T. Stolee, and Prof. René Just whose ideas, feedback, and contributions made this dissertation possible. I am also grateful to my colleagues and professors, Prof. Lori Clarke, Prof. Lee Osterweil, Heather Conboy,

Brittany Johnson, Emily First, Yixue Zhao, Alex Sanchez-Stern, Prof. Ingrid Holm, Prof. James Allan, and Prof. Brendan O'Connor for their feedback, and moral support throughout my PhD journey. Thanks should also go to Deborah Bergeron, Eileen Hamel, Kyle Skemer, Leeanne M. Leclerc, and Malaika Ross for helping me with the administrative work required to pursue my studies.

Many thanks to Dr. Smita Ghaisas, Prof. Jane Cleland-Huang, Prof. Suresh Purini, Prof. Vasudeva Verma, Prof. Kirti Garg, Preethu Rose, Nirav Ajmeri, Manoj Bhat, and Dilys Thomas, who provided me an exposure to the area of software engineering research, and inspired and supported me to pursue doctoral studies.

I would be remiss in not mentioning my father, Harish Motwani, and my mother, Neeta Motwani. Their belief in me has kept my spirits and motivation high during this process. Instead of providing me with just-enough education and making me join the family business, they supported me to pursue my super-ambitious childhood dream of becoming an engineer, a doctor, and a professor. I would also like to thank my sister, Neha Motwani Kukreja, and my brother-in-law, Govinda Kukreja, for supporting me in this journey by taking care of my parents and helping me focus on my studies without letting me worry while living thousands of miles away. I am deeply indebted to the struggles and sacrifices made by my family, and I am glad that finally, I can make them feel proud. I thank Yadav Sir, my high-school teacher, who taught me various career options in science and engineering and showed me the path to pursue higher studies. I would like to recognize Ashish Vishnoi, Anish Srivastava, Pankaj Agarwal, and Nirmal Singh for preparing me to compete with $500,000+$ students and secure a seat to study computer science and engineering at one of the top schools in India.

Finally, I thank my friends and cohort members, Ruchir Gupta, Vinay Garg, Sourabh Surana, Varun Goel, Karan Jindal, Anirudh Sabnis, Haresh Chudgar, Nishant Yadav, Aditya Tiwari, Raman Vaidya, Virat Shejwalkar, Niketh Murali, Sandhya Saisubramanian, Roopa Shenoy, Chase Lewis, Jessica Pavliska, Allan Rantala,

Helia Hashemi, Vinitra Ramasubramaniam, Satyanarayan Shukla, Sainyam Galhotra, Rico Angell, Ishita Zaman, Zhanna Kaufman, Utkarsh Agrawal, Abhinav Jangda, Donald Pinckney, Declan Gray-Mullen, Hamed Zamani, Justin Svegliato, Jane Tangen, Soha Rostaminia, Anthony Tracia, Mikhail Yasha, Seung Yeob Shin, Ravi Agrawal, Ravi Choudhary, Akansha Singh, Trapit Bansal, Raghavendra Addanki, Samuel Baxter, and Joseph Spitzer for helping me when in need and giving me unforgettable memories of pursuing this endeavor.

ABSTRACT

HIGH-QUALITY AUTOMATIC PROGRAM REPAIR

SEPTEMBER 2022

MANISH MOTWANI

B.Tech (Hons.), INTERNATIONAL INSTITUTE OF

INFORMATION TECHNOLOGY, HYDERABAD

M.S., UNIVERSITY OF MASSACHUSSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Yuriy Brun

Software developers spend significant time and effort fixing bugs. Automatic program repair promises to significantly reduce bug-fixing costs. Program repair requires: fault localization — identifying program elements that cause the bug, patch generation — identifying modifications to those program elements to attempt to repair the bug, and patch validation — verifying that the modification actually repairs the bug. Most automatic program repair techniques use the developer-written tests for the repair process and produce seemingly good patches for 11–19% of the bugs in real-world software. However, most of these patches are not correct, as they overfit to the developer-written tests and break undertested functionality. The goal of this dissertation is to address this patch overfitting problem.

We improve automatic program repair techniques by augmenting developer tests with additional constraints from natural language software artifacts. While most

existing techniques ignore such artifacts, we show that they can significantly improve the quality of program repair. We make the following contributions: (1) Methodologies to objectively evaluate repair techniques' repair applicability and repair quality; (2) Swami, a technique that uses natural language processing to improve the developer-written tests by generating executable tests with oracles from software specifications; (3) Blues, a technique that uses information-retrieval-based approach to identify suspicious program statements using bug reports; (4) RAFL, an unsupervised technique that uses rank-aggregation algorithms to combine multiple fault localization techniques; and (5) An evaluation demonstrating that automatic program repair can improve significantly when using both tests and bug reports, together.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

The global cost of software debugging has risen to $312 billion annually, and a significant amount of developers' time is spent on debugging and repairing software defects [91, 285]. Automatic program repair (APR) research (e.g., [19, 39, 42, 53, 54, 123, 142, 143, 145, 150, 174, 177, 180, 190, 239, 264, 269, 273, 296, 299]) aims to address this problem by devising techniques to automatically produce software patches to fix defects with minimal or no human intervention. The goal of APR techniques is to take a defective program and its specification (e.g., a suite of tests, some of which that program fails, or a set of formally specified constraints, some of which that program fails to satisfy) and produce a patched program that satisfies the specification. As developers typically write tests more often than formal specifications for their programs, APR techniques predominantly use developer-written test suites as program specifications to fix defects. This dissertation focuses on such test-suite-based APR techniques. Unfortunately, the patches produced by the test-suite-based APR techniques can repair some functionality encoded by the tests, while simultaneously breaking other, undertested functionality [252]. Thus, *quality* of the resulting patches is a critical concern, which prevents the wide-scale adoption of APR by practitioners. For example, companies such as Facebook and Bloomberg have only recently started to experiment using APR [126, 183, 249] to fix defects impacting their business. Recent results evaluating the quality of patches produced by APR techniques suggest that patch overfitting — patches that pass a particular set of test cases supplied to the program repair tool but fail to generalize to the desired

specification — is common [146, 176, 227, 252]. This makes developers lose trust in the APR techniques deterring their wide-scale adoption in practice [200]. The goal of this dissertation is to improve the quality of APR techniques.

This chapter is organized as follows. Section 1.1 provides a brief background on APR. Section 1.2 describes the three open research problems in APR that I address in this dissertation. Section 1.3 describes the five thrusts in which I divide this dissertation work along with the significance of the contributions made. Finally, Section 1.4 describes the outline of this dissertation.

1.1 Background: Automatic Program Repair

Test-suite-based APR techniques typically start with a program version and a suite of tests, some of which that program passes and some of which it fails, and then modify the program version until finding a set of modifications (a patch) that makes the program pass all the tests in that suite. Figure 1.1 shows the high-level program repair process that consists of the following three steps.

1. **Fault localization**: The goal of this step is to identify defective program elements (e.g., classes, methods, or statements) that cause the software defect. Automated fault localization (FL) used in APR typically uses static and runtime information about the program to identify program elements that may be the root cause of the defect.

2. **Patch generation**: This is the core algorithm of an APR technique. APR techniques use various algorithms to generate possible modifications that can be applied to the defective program elements that results in producing multiple candidate patches that could potentially fix the defect.

3. **Patch validation**: This step involves modifying a defective program by applying the automatically produced patch and validating its correctness against the

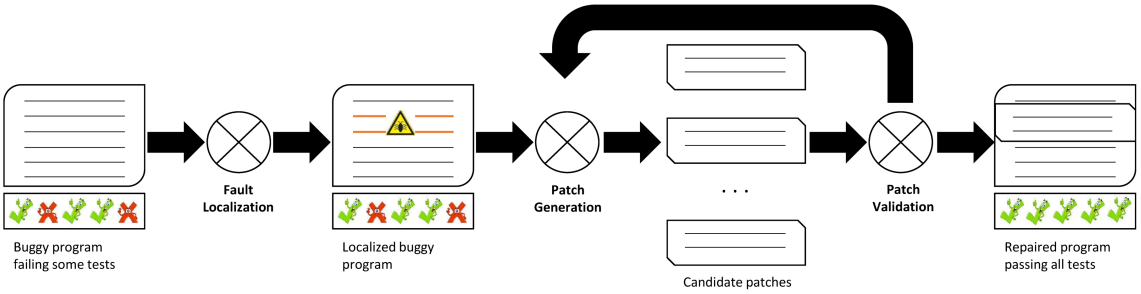


Figure 1.1: The three-step process of automatic program repair. The first step (*fault localization*) involves identifying program elements (e.g., classes, methods, or statements) that are causing the bug, the second step (*patch generation*) involves producing a patch (program modification) that when applied to the defective program elements would fix the defect, and the third step (*patch validation*) involves patching the defective program with a candidate patch and validating patched program against the test suite. If all tests pass, the candidate patch is reported as a repair and the process terminates otherwise, an attempt is made to produce new patch until the search space is exhausted or a timeout occurs.

developer-written test suite. If the patched program passes all the tests, the corresponding patch is reported as a *plausible* patch (patch that passes at least all tests used in the repair process). However, such a patch may overfit the tests and break other, under-tested functionality and therefore may not necessarily be a *correct* patch.

The method used for each of these steps can significantly affect the repair technique's success. Further, based on the kind of patch generation algorithm used, APR techniques can be categorized into the following three categories:

1. **Heuristic- and template-based repair techniques**. These repair techniques (e.g., GenProg [150], SimFix [111], Prophet [177], AE [282], HDRepair [145], ErrDoc [269], JAID [39], Qlose [53], and Par [123], among others) use certain heuristics or pre-defined templates to find a valid patch by iteratively exploring

a search space of candidate patches. For example, GenProg produces candidate patches by inserting, deleting, or replacing code at the abstract syntax tree level. SimFix mines code change operations from existing human-written patches and similar code snippets to build two search spaces. It then uses specific heuristics to produce candidate patches from the intersection of the two search spaces.

2. **Semantics-, synthesis-, and constraint-based repair techniques.** These repair techniques (e.g., Nopol [299], Semfix [204], DirectFix [189], Angelix [190], S3 [143], JFIX [142], etc.) use constraint solving and program synthesis to synthesize patches to satisfy semantics constraints extracted via symbolic execution and provided test suites. For example, Nopol repairs defective conditional statements by using the test cases to generate Satisfiability Modulo Theory (SMT) constraints that describe the desired program behavior on those test cases and uses an SMT solver to generate patched conditional expressions.

3. **Learning-based repair techniques.** These techniques (e.g., CURE [113], DeepFix [96], DeepRepair [289], SequenceR [40], CoCoNut [179], etc.) frame the program repair as neural machine translation (NMT) problem (translating defective program into patched program similar to translating one natural language into another) and use modern deep-learning-based algorithms to produce candidate patches. For example, CoCoNut uses a context-aware NMT architecture that represents the defective program and its surrounding context separately, to automatically fix defects in multiple programming languages. These techniques require additional training data (i.e., the tuples of defective program statements, context, and fixed program statements) to capture complex relations between defective and patched programs.

Existing research in APR has mostly focused on devising novel patch generation algorithms (e.g., heuristic-based [111,150,177,269,287], constraint-based [5,95,188,275],

and learning-based [41, 96, 244]) aimed to produce more correct patches. Recently, researchers have started investigating the effect of using different FL technologies, assumptions, and adaptations of FL techniques [11, 110, 132, 178, 263, 304], and patch validation methodologies [86, 268, 277, 297, 305, 311] on the performance of APR tools. Unlike these existing methods, the methods presented in this dissertation aim to improve APR by using information derived from natural-language software artifacts.

1.2 Problem Statement

Following are the three open research problems that I address in this dissertation.

1. **Evaluating Repair Applicability:** Improving APR requires knowing what kinds of defects APR tools are capable of patching. There exist many independent studies of multiple APR techniques that use different defect benchmarks that are all scattered. This problem aims to systematically reconcile these evaluations to get a clear picture of what kinds of defects APR techniques can patch and whether the patches produced are correct and acceptable to developers.

2. **Evaluating Repair Quality:** APR techniques produce many patches for defects in large, real-world programs with millions of lines of code and thousands of tests. However, studies show that even though these patches pass all the tests used in the repair process, most patches are incorrect and unacceptable to developers. Manually inspecting the correctness of these patches is infeasible and is subjected to manual bias, especially when the inspectors are authors of the repair tool who may not have a complete understanding of the program under repair. Because of this, developers lose trust in using off-the-shelf APR tools. This problem aims to develop an automated, objective, and scalable methodology to evaluate patch correctness that can provide more reliability to

the developers using APR tools and enable researchers to objectively compare the quality of multiple APR techniques.

3. **Improving Repair Quality:** As most patches produced by current APR techniques are incorrect or plausible, this problem aims to improve the quality of APR such that repair tools generate "correct" patches that are likely to be acceptable to developers. Although it might be infeasible to produce patches that are "guaranteed" to satisfy the developers' expectations, finding methods to generate and evaluate patches that are likely acceptable to developers is an open problem.

Addressing these problems is perhaps the most important step toward real-life adoption of APR. There are other open research problems in APR, such as extending APR to patch complex defects that repair tools currently struggle to patch and integrating APR into development workflow by ensuring reliability and without disrupting the workflow significantly [88, 151]. I plan to work on these research problems in the future.

1.3 Contributions

Most of the state-of-the-art program APR techniques use developer-written tests to: (a) localize the defect typically using spectrum-based fault localization techniques, which use the runtime information of passing and failing tests to localize the defective program elements and (b) generate and validate the automatically produced candidate patches based on the constraints imposed by the tests. While test suites provide an easy-to-use (because they are executable) specification, software typically contains many more artifacts that describe the desired *correct* software behavior. Many of these artifacts, such as requirements specifications, code comments, and bug reports use natural-language text to describe the bug and intended software behavior, and

are therefore not directly used in the repair process. I hypothesize that *if I can derive executable constraints from such artifacts and equip APR techniques with these additional constraints, it could further constraint the search space of the candidate patches and would improve the quality of patches produced.* The central goal of this dissertation is to test this hypothesis. For that, I divide my dissertation work into the following five thrusts:

1.3.1 Evaluating Repair Applicability (Chapter 2)

The goal of this thrust is to reconcile the scattered evaluations of program repair techniques to identify what kind of defects repair techniques can patch, and whether those defects are hard and important for developers. For this, I first develop an objective and scalable methodology to evaluate *repair applicability*, i.e., the characteristics of defects program repair techniques can patch. I then use my methodology to evaluate state-of-the-art APR techniques on real-world defects, and perform rigorous experiments to analyze the correlations between defect characteristics and the ability of repair techniques to produce patches.

1.3.2 Evaluating Repair Quality (Chapter 3)

The goal of this thrust is to develop an automated, objective, and scalable methodology to evaluate *repair quality*, i.e., the correctness of the patches produced by APR techniques, and identify the potential factors that affect repair quality. For this, I first develop a new methodology that uses *high-quality* held-out evaluation test suites to measure repair quality. I then use my methodology to evaluate the quality of state-of-the-art APR techniques on real-world defects. Further, I perform rigorous statistical analyses to analyze various factors that could affect repair quality.

1.3.3 Improving Developer-Written Tests Using Specifications (Chapter 4)

Most program repair techniques only use developer-written tests for the repair process. As developer-written tests are often incomplete and miss testing all the software functionality, it is one of the potential causes of producing patches that break other, under-tested functionality. The goal of this thrust is to improve the developer-written tests. I develop a technique that can automatically generate executable tests from natural language software specifications. I evaluate my technique by generating tests from the publicly accessible and reliable specifications of real-world software and analyze the effectiveness of the generated tests using two large, independently maintained, open-source software projects.

1.3.4 Improving Fault Localization Using Bug Reports (Chapter 5)

Modifying non-defective program statements placed before the defective ones in the fault localization output used by repair tools to construct patches is another potential cause of producing low quality patches. The goal of this thrust is to improve the accuracy of automated FL used in APR such that repair techniques are more likely to modify defective program statements for constructing candidate patches. For this, I first develop a new FL technique that localizes defects by ranking suspicious program statements using bug reports. Next, I develop an unsupervised technique to combine the results of multiple FL techniques that may use different bug information sources such as bug reports and test suites. I use my combining FL technique to combine test-suite-based and bug-report-based FL, and compare its effectiveness in localizing defects in large, real-world programs.

1.3.5 Improving Repair Quality Using Bug Reports and Tests Together (Chapter 6)

For this thrust, I put together my combined FL technique and test suites, and then use my repair quality evaluation methodology to test the hypothesis using real-world defects and state-of-the-art APR techniques.

Significance: APR has already shown effectiveness in real-world scenarios, but producing correct patches is one of the remaining hurdles preventing wide deployment in industry [88]. This dissertation makes progress towards addressing this challenge by developing (1) automated, objective, and scalable methods to evaluate APR along the dimensions of repair applicability and quality, (2) a new technique to improve test-suites used by APR techniques, (3) a new FL technique suitable for APR that uses bug reports to localize defects, (4) a new FL technique suitable for APR that combines information from bug reports and test suites demonstrating that it localizes defects better than underlying techniques that use only bug reports or only tests, and (5) demonstrating that with this new FL, APR techniques can patch more defects correctly. This dissertation presents the first APR techniques that use both bug reports and tests.

1.4 Dissertation Outline

This dissertation is structured as follows. Chapter 2 describes a study to evaluate APR applicability. Chapter 3 describes a study to evaluate APR quality, and analyzing factors that could affect repair quality. Chapter 4 describes a technique to generate executable tests with oracles from natural-language software specifications. Chapter 5 describes techniques to improve FL using bug reports. Chapter 6 describes how using both tests and bug reports improves APR. Chapter 7 places the presented work in the context of related research. Chapter 8 summarizes the contributions, and presents future directions for the research.

CHAPTER 2

APPLICABILITY OF AUTOMATIC PROGRAM REPAIR ON REAL-WORLD DEFECTS

2.1 Introduction

APR techniques have been evaluated in terms of how many defects they produce a patch for (e.g., [122, 123, 147, 227]), how quickly they produce patches (e.g., [147, 282]), and the quality of the patches they produce, such as developer-judged correctness [177, 184, 227], how many independent tests the patched programs pass [29, 112, 252], how maintainable the patches are [83], and how likely developers are to accept them [123]. However, prior work has not studied the defect characteristics that make APR more applicable. Researchers have stressed the need to identify the classes of defects for which repair techniques work well [195].

In this chapter, we present a methodology to evaluate the kinds of defects for which APR techniques produce patches, answering the question of whether existing techniques can patch important defects, or defects that are hard for developers to repair. The same way work on evaluating patch quality [29, 184, 216, 227, 252] has led to work on improving repair quality [122, 175, 177], this work can lead to work on improving the applicability of APR to patch defects that they currently struggle to patch.

This chapter is organized as follows. Section 2.2 describes a methodology to characterize defects' importance and difficulty followed by the statistical tests to measure the association between defect characteristics and the ability of repair techniques to patch defects. Section 2.3 describes the repair techniques and defect

benchmarks we use for applying our methodology. Section 2.4 describes the evaluation and key findings in terms of the research questions we ask and Section 2.5 discusses their implications. Section 2.6 addresses the threats to validity of our study and Section 2.7 summarizes our contributions.

2.2 Characterizing Defect Importance and Difficulty

Classifying how difficult a defect is to repair, and how important repairing a defect is to a project is a complex and subjective task. There is neither a single measure of difficulty nor of importance. To identify aspects of defects related to difficulty and importance, we first analyzed eight popular bug tracking systems [254], three popular open-source project hosting platforms with bug tracking systems, and two benchmarks of software defects (that include source code, test suites, and developer-written patches) [119, 149].

We used constructivist grounded theory [31] with coding and constant comparison that is specifically designed for reasoning about and categorizing concepts without preconceived abstractions of the involved data [37]. In other words, we started out without having strong preconceived notions of what data found in bug tracking systems, open-source project hosting platforms, and defect benchmarks are likely to be relevant to defect difficulty and importance, and we used the appropriate grounded theory for identifying such data and classifying them into abstractions. This methodology has been previously recommended for use in information systems research [187]. Two of the researchers, called coders, independently analyzed all *concrete parameters* available in the bug tracking systems, open-source project hosting platforms, and the defect benchmarks (specifically focusing on the test suites and developer-written patches available in these benchmarks). The coders selected which pieces of data may be relevant to how difficult or important the defect is to repair. For example, concrete parameter *priority* was associated with the importance of defect while *# of lines in the*

minimized patch for a defect was associated with difficulty. Coders also identified other data such as number of triggering and relevant tests available for a defect and number of project versions affected by a defect that they felt may be interesting to correlate with automated repair techniques' ability to repair the defect. The two coders then compared their coding and reconciled the differences. Reconciling sometimes required looking at example defects to gain a further insight into the semantics of the concrete parameters.

Next, the coders, again independently, grouped similar concrete parameters (e.g., identical parameters for which different bug tracking systems use different names, or closely related concrete parameters) into *abstract parameters*. For example concrete parameters such as *components*, *linked entities*, *affects versions* and *fix versions* were grouped together to form an abstract parameter *versions*. Again, the coders reconciled their coding. The coders iterated between identifying concrete and abstract parameters until their findings saturated and no more parameters were identified. At the end of analysis, we had created eleven abstract parameters.

Finally, the coders, again independently, categorized the abstract parameters by grouping closely related parameters, and then reconciled their coding. We call these categories *defect characteristics*. For example, abstract parameters *File count*, *Line Count* and *Reproducibility* were grouped together to form a defect characteristic *Complexity*. Similarly abstract parameters *Statement coverage*, *Triggering test count* and *Relevant test count* were grouped together to form the defect characteristic *Test-Effectiveness*. We came up with five such defect characteristics using eleven abstract parameters.

The eight popular bug tracking systems we used are Bugzilla, JIRA, IBM Rational ClearQuest, Mantis, Trac, Redmine, HP ALM Quality Center, and FogBugz [254]. The three popular open-source project hosting platforms with bug tracking systems we used are Sourceforge, GitHub, and Google code (although the latter is no longer

active). Finally, the two benchmarks of software defects we used are a 185-C-defect ManyBugs [149] and a 357-Java-defect Defects4J [119] benchmarks. For completeness and reproducibility of our research, we include a complete list and description of the concrete parameters we found for each of the bug tracking systems, project-hosting platforms, and defect benchmarks in Figure A.1 in Appendix A. Note that the names of the parameters bug tracking systems use are not always intuitive, and sometimes inconsistent between systems. For example, Google code uses the terms "open" and "closed" for timestamps of when an issue was open or closed. GitHub uses these terms as binary labels. Google code uses the parameter "status" to encode these labels. We do not include a detailed description of what information each parameter encodes and how it encodes it, but this information is available from the underlying bug tracking systems and project-hosting platforms. The complete mapping of concrete parameters to abstract parameters, and, in turn, to the defect characteristics is shown in Figure A.2 in Appendix A.

Sections 2.2.2–2.2.6 describe the five defect characteristics and the eleven abstract parameters that map onto them. But first, Section 2.2.1 describes the statistical tests we use to determine whether the abstract parameters correlate with *repairability* — the APR techniques' ability to produce patches.

2.2.1 Statistical Tests

Ten out of the eleven abstract parameters are numerical. For most parameters, the number of unique values is small and the magnitude of their difference may not be indicative. Therefore, we rely on non-parametric statistics and do not assume that the underlying values of a parameter should be interpreted as equidistant from one another. Specifically, for each technique and for each of the ten numerical abstract parameters, we split the distribution of that parameter's values into two distribution samples: (1) the distribution of the parameter's values for the defects for which the

technique produces a patch, and (2) the distribution of the parameter's values for the defects for which the technique does not produce a patch. We use the non-parametric Mann-Whitney U test to determine if the two distribution samples are statistically significantly different. That is, the test computes a p value that indicates whether the null hypothesis that the two distribution samples are statistically indistinguishable (i.e., they are drawn from the same distribution) should be rejected. We use the standard convention of $p \leq 0.01$ to mean the difference is strongly statistically significant, $0.01 < p \leq 0.05$ to mean the difference is statistically significant, $0.05 < p \leq 0.1$ to mean the difference is weakly statistically significant, and $p > 0.1$ to mean the difference is not statistically significant. We measure the strength of the association between the abstract parameter and the technique's ability to produce a patch using the rank-biserial correlation coefficient, a special case of Somers' d. A defect's repairability with respect to a technique — whether the technique produces a patch for this defect — is a dichotomous variable, and in such situations, Somers' d is a recommended measure of the non-parametric effect size for ordinal data; Somers' d is also asymmetric with the presumed cause and effect variables, which is the case in our study [79, 203]. We use the standard mapping from Somers' d (which can take on values between -1 and 1) to the adjectives very weak ($|d| < 0.1$), weak ($0.1 \leq |d| < 0.2$), moderate ($0.2 \leq |d| < 0.3$), and strong ($0.3 \leq |d|$) [152]. We further compute the 95% confidence interval for Somers' d (referred to as 95% CI). We consider an association to be statistically and practically significant if it is at least weakly statistically significant and if the 95% CI for Somers' d is entirely positive or entirely negative. For the eleventh abstract parameter, developer-written patch characteristics, we use a logistic regression to fit a model for repairability and determine which characteristics have the strongest effect on repairability (Section 2.2.6).

2.2.2 Defect Importance

Our analysis of the eleven issue tracking systems (eight bug trackers and three project-hosting platforms) identified three common abstract parameters related to importance: priority, project versions affected, and time to fix the defect.

Priority of the defect. The priority of a defect is included in nine out of the eleven issue tracking systems. Different issue tracking systems use different ordinal scales to measure the priority. We mapped these different scales to our own scale that varies from 1 (lowest priority) to 5 (highest priority). Our study determines if there is a significant association between priority and repairability using the non-parametric Mann-Whitney U test and measures the strength of the association using a rank-biserial correlation coefficient Somers' d. In other words, as described in Section 2.2.1, we compare the distribution of priorities of defects patched by an APR technique to the distribution of priorities of defects not patched by an APR technique. We use the Mann-Whitney U test to test if the distributions are statistically significantly different, and we measure the magnitude of that difference using Somers' d.

Does the defect affect more than one project version? The number of project versions a defect affects is included in three out of the eleven issue tracking systems. Our study uses the Mann-Whitney U test to check for a significant association with repairability, and it measures the strength of the association using Somers' d.

Time taken to fix the defect. The time between when a defect was reported and when it was resolved is included in eight out of the eleven issue tracking systems. For those systems that did not have any concrete parameters to indicate the time when defect was resolved, we approximated it by computing the time between timestamps of when the issue was entered into an issue tracking system, and the last commit for the issue. Our study determines if there is a significant association between time to fix and repairability using the Mann-Whitney U test, and it measures the strength of the association using Somers' d. Note that time to fix a defect could be considered

as a parameter for importance or difficulty, but based on our analysis of defects and experience of commits made by developers, associating this parameter to the importance characteristic seem more accurate.

2.2.3 Defect Complexity

Our study considers two defect complexity measures: number of files edited by the developer-written patch, and number of non-blank, non-comment lines of code in that patch. These patches are manually minimized in the Defects4J benchmark to remove all changes that do not contribute to the patch's goal [119], but are not in the ManyBugs benchmark. We partially minimized the ManyBugs patches by removing all blank and comment lines to reduce the potential bias due to over-approximating the number of files and lines of code. We employed the `diffstat` tool to automatically compute the number of source code lines affected by the partially minimized patches.

Number of source files edited by the developer-written patch. A defect that requires editing multiple source files might be harder to localize and generally more difficult to repair. Our study investigates the effect of the number of source files edited by a patch on a defect's repairability. It determines if there is a significant association between the number of files edited and repairability using the Mann-Whitney U test. It measures the strength of the association using Somers' d.

Number of non-blank, non-comment lines of code in developer-written patch. Our study also investigates if defects with larger developer-written patches, in terms of lines of code, are more difficult for automated repair techniques to repair. Our study determines whether the number of lines of code has a significant association with repairability using the Mann-Whitney U test, and it measures the strength of the association using Somers' d.

2.2.4 Test Effectiveness

Prior work [252] suggests that test effectiveness might have an effect on an automated repair technique's ability to generate a patch. Our study identified three parameters related to test effectiveness: (1) the fraction of the lines in the files edited by the developer-written patches that are executed by the test suite, (2) the number of defect-triggering test cases, and (3) the number of relevant test cases (test cases that execute at least one line of the developer-written patch). Our study determines, for each parameter, if it has a significant association with repairability using the Mann-Whitney U test, and it measures the strength of the association using Somers' d.

2.2.5 Defect Independence

Our analysis of eleven issue tracking systems identified dependents as a common abstract parameter related to difficulty: a defect whose repair depends on another issue in the issue tracking system might be more difficult to repair than a defect that can be repaired independently. The information about defect dependents is included in five out of the eleven issue tracking systems. Our study uses the Mann-Whitney U test to check for a significant association with repairability, and it measures the strength of the association using Somers' d.

2.2.6 Developer-Written Patch Characteristics

The ManyBugs benchmark provides characteristics of the developer-written patches for its defects [149]. The nine characteristics describe if the developer-written patch:

C1: changes one or more data structures or types

C2: changes one or more method signatures

C3: changes one or more arguments to one or more functions

C4: adds one or more function calls

C5: changes one or more conditionals

C6: adds one or more new variables

C7: adds one or more if statements

C8: adds one or more loops

C9: adds one or more new functions

We consider each patch characteristic as a dichotomous variable and use a logistic regression to fit a model for repairability. It determines the patch characteristics that have the strongest effect on repairability.

2.3 Subjects of Investigation

This section details the subjects we investigated, in terms of the automated repair techniques and the real-world defects characterized using the five defect characteristics described in Sections 2.2.2–2.2.6.

2.3.1 Automated Program Repair Techniques

Our study considers seven state-of-the-art APR techniques.

GenProg [147, 150, 284] is a heuristics-based repair technique. Such techniques create candidate patches, often using search-based approaches [97], and then validate them, typically through testing. GenProg uses a genetic programming heuristic [133] to search through the space of possible patches, mutating lines executed by failing test cases, either deleting them, inserting lines of code from elsewhere in the program, or both to create new potential patches, and crossover operators to combine patches. GenProg uses the test suite to select the best-fit patch candidates and continues evolving them until it either finds a patch that passes all tests, or until a specified timeout. GenProg targets general defects without focusing on a specific class. GenProg

was originally designed for C programs [147, 150, 284], but has been reimplemented for Java [184]. This study differentiates these two implementations as GenProgC and GenProgJ.

TrpAutoRepair [224] (also published under the name RSRepair in "The strength of random search on automated program repair" by Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang in the 2014 International Conference on Software Engineering; we refer to the original name in this study) uses random search instead of GenProg's genetic programming to traverse the search space of candidate patches for C programs. It uses heuristics to select the most informative test cases first, and stops running the suite once a test fails. TrpAutoRepair limits its patches to a single edit. It is more efficient than GenProg in terms of time and test case evaluations [224]. Similarly to GenProg, TrpAutoRepair targets general defects without focusing on a specific class.

AE [282] is a deterministic repair technique that uses heuristic computation of program equivalence to prune the space of possible repairs, selectively choosing which tests to use to validate intermediate patch candidates. AE uses the same change operators as GenProg and TrpAutoRepair, but rather than using a genetic or randomized search algorithm, AE exhaustively searches through the space of all non-equivalent k-distance edits. AE targets C programs, and, again, targets general defects without focusing on a specific class.

Kali [227] is a simple heuristics-based repair technique that only deletes lines of code. It was originally designed to show that even this simple approach can sometimes produce patches that pass the available tests, but it has been shown that at times, these patches are of high quality [184, 227]. Kali was originally designed for C programs [227], but has been reimplemented for Java [184]. This study differentiates these two implementations as KaliC and KaliJ. Kali targets defects that can be patched strictly by removing functionality.

SPR (staged program repair) [175] is a heuristics-based repair technique that uses a set of predefined, parameterized transformation schemas designed to generate repairs for specific defect classes. SPR targets defects that can be repaired by inserting or modifying conditional statements, initializing variables, replacing one variable with another, replacing one invoked function with another, replacing one constant with another, or inserts a statement from elsewhere in the program. Many of the schemas generate conditions by first computing constraints over specific variable values needed for a repair and then synthesizing logical expressions to satisfy those constraints. SPR uses the test suite to validate the patches, targets C programs, and has been shown to find higher-quality patches than GenProg [175].

Prophet [177] explores single-edit potential patches, similarly to TrpAutoRepair, and SPR's transformation schemas. It prioritizes the schemas using models inferred from successful developer-written patches from open-source development. The class of defects Prophet targets is the same as SPR's. Prophet targets C programs and has been shown to find higher-quality patches than GenProgC, AE, SPR, and KaliC [177].

Nopol [59] is a synthesis-based technique that targets repairing conditional statements in Java programs. Using the test cases, Nopol generates Satisfiability Modulo Theory (SMT) constraints that describe the desired behavior on those test cases and uses an SMT solver to generate a conditional. Nopol fixes defects such as forgotten null pointer checks.

Repairability Information: ManyBugs and Defects4J benchmarks (described next in Section 2.3.2) have been used to evaluate APR in the past [149, 150, 175, 177, 184, 227]. GenProgC, TrpAutoRepair, and AE have been applied to all 185 ManyBugs defects [149]. SPR, Prophet, and KaliC have each been applied to a 105-defect subset of the 185-defect ManyBugs benchmark [175, 177, 227]. Nopol, GenProgJ, and KaliJ have been applied to 224 Defects4J defects [184]. In our evaluation, we use these results for the 409 defects, in terms of which techniques can produce a patch for which

defects. Figure 2.1 summarizes these results. Combined, the techniques repair 56% and 49% of C defects from the two sets of C defects and 21% of the Java defects, denoted by the "⋃C" and "⋃Java" rows. We believe the difference in these numbers is due in part to the fact that we study more techniques that target C. Additionally, the first C-targeting techniques [284] predate the Java-targeting ones, and the first versions of ManyBugs [147] predate the first version of Defects4J [119], so researchers have had more time to improve their techniques for ManyBugs than for Defects4J.

Further, research studying the quality of repair has identified, via manual analysis and judgment, which subset of the defects have correct patches (and which are "plausible but incorrect") for GenProgC, TrpAutoRepair, and AE [227], SPR, Prophet, and KaliC [177], and GenProgJ, Nopol, and KaliJ [184]. We use these data to analyze how defect characteristics correlate with repair quality. Because far fewer of the defects are automatically repaired correctly than plausibly, we expect the statistical power of these measurements to be less significant.

2.3.2 Defect Benchmarks

We use two benchmarks of defects for this study, ManyBugs and Defects4J.

The **ManyBugs** benchmark [149] consists of 185 defect scenarios, summarized in the top of Figure 2.2. Each scenario consists of a version of source code from one of nine large, open-source software systems, a set of project tests that fail on that version, a set of tests that pass on that version, and another version from a later point in the repository that passes all the tests. These defect properties allow for some defects to actually be features that modify system behavior. Out of 185 defects in ManyBugs, 29 defects were features and the remaining 156 defects were bugs. For 122 of the defects, ManyBugs includes accurate links to the project's bug tracking system or forums (though 8 of those links are no longer accessible), describing the defect.

technique	patched	unpatched	patched %
ManyBugs			
GenProgC	87	98	47%
TrpAutoRepair	97	88	52%
AE	86	99	46%
⋃C	**104**	**81**	**56%**
ManyBugs (105-defect subset)			
KaliC	27	78	26%
SPR	46	59	44%
Prophet	43	62	41%
⋃C	**52**	**53**	**49%**
Defects4J			
GenProgJ	27	197	12%
KaliJ	22	202	10%
Nopol	35	189	16%
⋃Java	**47**	**177**	**21%**

Figure 2.1: Automated repair techniques we use to study repair applicability. The techniques that have been evaluated on the entire 185-defect ManyBugs patched 56% of those C defects; the techniques that have been evaluated on the 105-defect subset of ManyBugs patched 49% of those C defects; the techniques that have been evaluated on 224-defect Defects4J patched 21% of those Java defects. The "⋃C" and "⋃Java" rows give the numbers and ratios of defects for which at least one of the C- and Java-targeting techniques could generate a patch.

The **Defects4J** benchmark [119] consists of 357 defect scenarios, summarized in the bottom of Figure 2.2. Similarly to ManyBugs, each scenario includes a version of source code with a defect, and a version with that defect repaired by a developer. The benchmark also includes, for each defect, a developer-written test suite that includes at least one triggering test. As the repairability information in Section 2.3.1 describes, 224 of these defects have been used for APR, and so we consider that 224-defect subset. Of these, 205 have links to the project's bug tracking system. We found that 4 out of the 224 defects were features and the remaining 220 defects were bugs.

Characterizing the ManyBugs and Defects4J Data: The defect benchmarks we study provide partial information for some of the defects, e.g., only some of the defects contain a link to an issue in the issue tracking system. We next describe the

ManyBugs			
program	kLoC	defects	program description
`fbc`	97	3	legacy language compiler
`gmp`	145	2	multi-precision math library
`gzip`	491	5	data compression utility
`libtiff`	77	24	image processing library
`lighttpd`	62	9	web server
`php`	1,099	104	web programming language
`python`	407	15	general-purpose language
`valgrind`	793	15	dynamic debugging tool
`wireshark`	2,814	8	network packet analyzer
total	**5,985**	**185**	
Defects4J			
`JFreeChart`	96	26	chart drawing library
`Closure`	90	133	compiler
`Commons Lang`	22	65	Apache core library
`Commons Math`	85	106	Apache math and stat library
`Joda-Time`	28	27	date and time library
total	**321**	**357**	

Figure 2.2: Defect benchmarks used to study repair applicability. We use the ManyBugs and Defects4J benchmarks. The 185 ManyBugs defects come from nine open-source software systems [149], and the 357 Defects4J defects come from five open-source software systems [119]. The `Closure` defects are excluded from our study because prior studies have not used them to evaluate automated repair techniques [184].

information we were able to obtain for these defects, to approximate the idealized methodology described in Section 2.2.

Recall that each defect in the ManyBugs and Defects4J benchmarks corresponds to a pair of commits in a version control system, but not necessarily to an issue in an issue tracking system. For each defect in Defects4J, we tried to manually determine the corresponding issue in the issue tracking system by cross-referencing the commit logs and commit IDs with the commit information in the issue tracking system. For ManyBugs, the information about the issues in the issue tracking system which are associated with a defect was available for 122 out of the 185 defects (though 8 are no longer accessible because either the URL did not resolve or the issue was private). For

Defects4J, this information was available for 205 out of the 224 defects. We annotated each defect with a link to the issue tracking system, with the abstract parameters recorded in the issue tracking system. The abstract parameters recorded were obtained from different concrete parameters depending on the issue tracking system used by a given project. Also, information about some of the abstract parameters was not found in some of the issue tracking systems. Hence we couldn't annotate all the defects with all the abstract parameters. Figure B.1 in Appendix B shows the details about information available for annotating the defects with parameters obtained from issue tracking systems.

For the abstract parameters that were obtained from the two defect benchmarks, we were able to annotate all defects with *line count*, *file count*, *triggering test count*, and *relevant test count* as this information was available with the benchmarks. The triggering test count is the number of negative tests for a defect provided in ManyBugs and number of triggering tests for a defect provided in Defects4J. The relevant test count in Defects4J is the number of test cases that execute at least one statement in at least one file edited by the developer-written patch. These are provided as relevant tests for each defect in Defects4J. ManyBugs provides all test cases that are relevant for the project, but these may not be specific to patched file(s). The relevant test count for ManyBugs is the number of all tests relevant for the project.

We annotated each defect in Defects4J with the statement coverage ratio of the test suite on the file(s) edited by the developer-written patch, using the `coverage` utility provided by the Defects4J framework. For ManyBugs, we used the `gcov` tool to compute this information for all the defects except for 52 defects that we could not compile. Figure B.2 in Appendix B shows the number of defects that could be annotated with each abstract parameter.

While analyzing the defects in ManyBugs and Defects4J, we found that some of the defects were actually features. We classified all the defects and found that 29

out of the 185 ManyBugs defects were features while the remaining 156 were bugs, and that 4 out of the 224 Defects4J defects were features while the remaining 220 were bugs. To make this classification, we manually analyzed the issue description and discussion in the issue tracking system (when this information was available) to identify if the issue directly related to implementing new functionality, extending or enhancing functionality, or conforming to a standard. For those issues that satisfied that criterion, we then checked if the issue related to an unexpected output; if it did, we classified it as a bug. We then analyzed the minimized, developer-written source code changes made to resolve the issue for the issues not already classified as bugs to verify that the changes were consistent with the issue description and discussion, leading to the final classification as a feature. For issues without links to an issue tracking system, we followed the same procedure using the developer-written log messages and source code changes.

We considered two potential confounding factors that could affect repairability: (1) the defect type (if the defect relates to a bug report or a feature request), and (2) whether a defect links to an issue in an issue tracking system. The purpose of this analysis was to determine if our study needs to control for these factors. We used Fisher's exact test to test for independence. Figure 2.3 shows that the repairability results are not independent of the defect type for ManyBugs, and hence our study controls for this factor by analyzing bug reports and feature requests separately. Fisher's exact test confirmed that there is no significant association between repairability and whether a defect links to an issue in an issue tracking system ($p = 0.64$ for ManyBugs; $p = 1.0$ for the Defects4J subject Chart, the only subject in Defects4J with some missing issue links). Therefore, our study does not control for this factor.

Our study uses the complexity of the developer-written patches as a proxy for defect complexity, instead of inferring complexity from the issue tracking systems

ManyBugs			
defect type	patched	unpatched	total
bug	105	51	156
feature	14	15	29
column total	119	66	185
Fisher's exact test: $p = 0.05$			

Defects4J			
defect type	patched	unpatched	total
bug	46	174	220
feature	1	3	4
column total	47	177	224
Fisher's exact test: $p = 1.00$			

Figure 2.3: The effect of a defect's type (bug or feature) on its repairability.

for two reasons. First, a developer-written patch is available for every defect in the ManyBugs and Defects4J benchmarks — by contrast, only 327 out of 409 defects have a corresponding issue in an issue tracking system. Second, the defect complexity recorded in an issue tracking system may be subjective or specific to the project. Note that while a defect might, in theory, have an unbound number of valid fixes, we assume that the developer-written fix is indicative of the complexity of the defect it fixes.

Benchmark extensions. As part of this work, we augmented ManyBugs and Defects4J with extra information. We annotated every defect with the number of lines of code in the minimized, developer-written patch, the number of files that patch touches, the number of relevant test cases (test cases that execute at least one statement in at least one file edited by the developer-written patch) as well as test cases that trigger the defect and, the test suite coverage. We have also annotated the 114 ManyBugs and 205 Defects4J defects with links to their projects' bug tracking systems or forums with how much time passed between when the defect was reported and when it was resolved, the priority of the defect, the number of project versions impacted by the defect, and the number of dependent defects. These annotations enable our evaluation, and can enable others to evaluate how defect characteristics correlate

with the applicability of their repair techniques. The annotations are available at `https://github.com/LASER-UMASS/AutomatedRepairApplicabilityData/`.

2.4 Evaluating Repair Applicability and Key Findings

We consider nine automated repair tools that implement seven APR techniques (recall Section 2.3.1) (two pairs of tools implement the same technique for different languages). Six of the techniques repair C programs: GenProg [147, 150, 284], TrpAutoRepair [224], AE [282], Kali [227], SPR [175], and Prophet [177]. Three of the techniques repair Java programs: Nopol [59], a Java reimplementation of GenProg [184], and a Java reimplementation of Kali [184]. We use results from prior evaluations of these techniques on the ManyBugs and Defects4J datasets. GenProg, TrpAutoRepair, and AE have been applied to all 185 ManyBugs defects [149]. SPR, Prophet, and Kali have each been applied to 105 (a strict subset of the 185) ManyBugs defects [175, 177, 227]. Nopol, and the Java versions of GenProg and Kali have been applied to 224 Defects4J defects [184].

We identify and compute eleven unique abstract parameters recorded in most bug tracking systems and source code repositories that relate to five defect characteristics: importance, independence, complexity, test effectiveness, and characteristics of the developer-written patch. These parameters and characteristics form the basis of our evaluation, comprising the dataset and methodology that creators of new Java and C automated repair tools can use to evaluate their tools. We use the statistical tests described in Section 2.2.1 to determine whether the abstract parameters correlate with the automated repair techniques' ability to produce patches.

Our evaluation answers six research questions:

RQ1 *Importance*: Is an APR technique's ability to produce a patch for a defect correlated with that defect's importance?

Answer: Java repair techniques are moderately more likely to produce patches for defects of a higher priority, while C repair techniques' ability to produce a patch for a defect does not correlate with defect priority. Further, Java and C repair techniques' ability to produce a patch for a defect has little to no consistent correlation with the time taken by developer(s) to fix that defect and the number of software versions affected by that defect. This suggests that automated repair is as likely to produce a patch for a defect that takes developers a long time to fix as for a defect that developers fix quickly.

RQ2 *Complexity*: Is an APR technique's ability to produce a patch for a defect correlated with that defect's complexity?

Answer: C repair techniques are less likely to produce patches for defects that required developers to write more lines of code and edit more files to patch. However, the observed negative correlations are not consistently strong for all techniques, suggesting that automated repair can still produce patches for some complex defects. Further, for Java repair techniques, we do not observe a statistically significant relationship of this kind.

RQ3 *Test effectiveness*: Is an APR technique's ability to produce a patch for a defect correlated with the effectiveness of the test suite used to repair that defect?

Answer: Java repair techniques are less likely to produce patches for defects with more triggering or more relevant tests, while C repair techniques' ability to produce a patch for a defect does not correlate with the number of triggering or relevant tests. Further, Java and C repair techniques' ability to produce a patch for a

defect has little to no consistent correlation with the statement coverage of the test suite used to repair that defect.

RQ4 *Independence*: Is an APR technique's ability to produce a patch for a defect correlated with that defect's dependence on other defects?

Answer: Java repair techniques' ability to produce a patch for a defect does not correlate with that defect's dependence on other defects. For the C repair techniques, the data does not provide sufficient diversity to study the relationship between repairability and defect independence.

RQ5 *Characteristics of developers' patches*: What characteristics of the developer-written patch are significantly associated with an APR technique's ability to produce a patch for that defect?

Answer: Java and C repair techniques struggle to produce patches for defects that required developers to insert loops or new function calls, or change method signatures.

RQ6 *Patch quality*: What characteristics of defect are significantly associated with an APR technique's ability to produce a high-quality patch for that defect?

Answer: Only two of the considered repair techniques, Prophet and SPR, produce a sufficient number of high-quality patches to evaluate. These techniques were less likely to produce patches for more complex defects, and they were even less likely to produce correct patches.

In the following sections, we describe the details about the results obtained for each of the above research questions. Sections 2.4.1–2.4.6 present results for our six research questions, Section 2.4.7 comments on feature synthesis.

2.4.1 Defect Importance

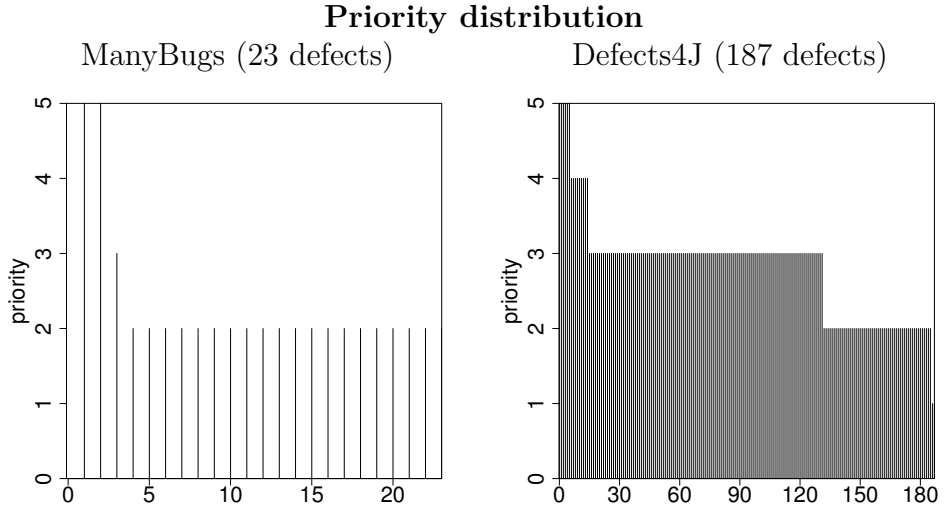
RQ1: Is an APR technique's ability to produce a patch for a defect correlated with that defect's importance?

We measure a defect's importance using the defect's *priority* value in the project's issue tracking system, the number of project versions the defect affects, and the time from when the defect was reported until it was resolved (recall Section 2.2.2).

Priority of the defect. For 156 defects in ManyBugs that were classified as bugs, 23 defects had priority values: 20 have priority two, 1 has priority three, and 2 have priority five. For 220 defects in Defects4J that were classified as bugs, 187 defects had priority values: 2 have priority one, 54 have priority two, 117 have priority three, 9 have priority four, and 5 have priority five. Top of Figure 2.4 shows the distribution of the priority values.

Figure 2.4 shows the results of the Somers' d and the Mann-Whitney U tests comparing the priority distributions of defects for which techniques do, and do not produce patches. For the Java techniques, Somers' d indicates moderate to strong positive correlations between priority and repairability. For Nopol and $\bigcup$Java, the Mann-Whitney U test indicates a statistically significant difference between the distributions ($p \leq 0.05$) and the Somers' d 95% CI is entirely positive. While we observed a weakly significant, moderate positive correlation ($p \leq 0.1$) for GenProgJ and KaliJ, our confounding factor analysis (Section 2.5.3) suggests that this observation was due to a correlation between the priority and the number of relevant test cases.

For C techniques, we observe weak to moderate, both positive and negative correlations. Because relatively few (23) of the C defects have priority values, the Mann-Whitney U test does not find any statistically significant differences between the priority distributions of defects for which techniques do, and do not produce patches. Therefore we make no claims about a correlation between priority and C techniques' ability to produce patches.



technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	0.100	[−0.299, 0.499]	0.856	5	18
TrpAutoRepair	0.167	[−0.147, 0.480]	0.348	9	14
AE	−0.176	[−0.341, −0.012]	0.539	6	17
⋃C	0.114	[−0.165, 0.392]	0.342	11	12
ManyBugs (105-defect subset)					
SPR	−0.111	[−0.541, 0.319]	0.753	9	5
KaliC	−0.111	[−0.541, 0.319]	0.753	9	5
Prophet	−0.222	[−0.472, 0.027]	0.505	5	9
⋃C	−0.175	[−0.679, 0.329]	0.780	10	4
Defects4J					
GenProgJ	0.206	[−0.037, 0.449]	0.088	20	167
Nopol	**0.307**	[**0.133**, **0.481**]	**0.002**	**28**	**159**
KaliJ	0.223	[−0.023, 0.468]	0.080	16	171
⋃Java	**0.216**	[**0.065**, **0.367**]	**0.017**	**37**	**150**

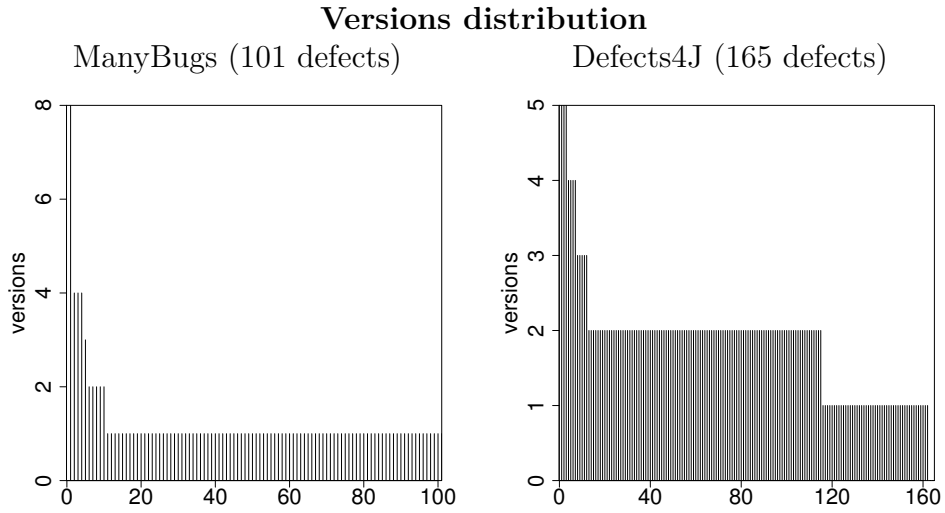
Figure 2.4: Correlation between defect priority and repair applicability. Priority data are available for 23 ManyBugs and 187 Defects4J defects classified as bugs. Java repair techniques are more likely to produce a patch for defects with a higher priority. Insufficient data for C defects prevent a statistically significant conclusion. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are **bold**. The data shown are only for defects classified as bugs and with known priority values.

Does the defect affect more than one project version? For 156 defects in ManyBugs that were classified as bugs, 101 defects had information on how many

versions they affect. Of these, 91 affected a single version, 5 affected two versions, 1 affected 3 versions, 3 affected four versions, and 1 affected eight versions. For 220 defects in Defects4J that were classified as bugs, 165 defects had this information. Of these, 50 affected a single version, 103 affected two versions, 5 affected three versions, 4 affected four versions, and 3 affected five versions. Top of Figure 2.5 shows the distribution of the versions values.

Figure 2.5 shows the results of the Somers' d and the Mann-Whitney U tests on the versions distributions of defects for which techniques do, and do not produce patches. We found no evidence of a relationship between a defect's repairability and the number of versions it depends on except for GenProgC and AE, which showed a significant ($p \leq 0.05$) and weakly significant ($p \leq 0.1$), respectively, negative correlation with the number of versions affected by a defect. For all other techniques, the results were insignificant ($p > 0.1$ or the 95% CI spanned zero). We conclude that the number of versions affected by a defect likely has negligible effect on automated repair's effectiveness producing a patch for that defect.

Time taken to fix the defect. For 156 defects in ManyBugs that were classified as bugs, 95 defects had information about the time frame that passed between when the defect was reported and when it was resolved. This time to fix the defect varied from 43 minutes to 10.7 years. Out of these 95 defects, 48 have time to fix of less than one month, 32 from one month to one year, and 15 greater than one year. For 220 defects in Defects4J that were classified as bugs, 199 had this information. The time to fix varied from 1 minute 21 seconds to 4.0 years. Out of 220 defects, 140 have time to fix of less than one month, 42 from one month to one year, and 17 greater than one year. The defects with a low time to fix may exemplify a source of potential noise in our data. While it is rare for the developer(s) to repair a defect in a minute and a half, they will sometimes discover a defect, think about the correct way to repair it, and even write relevant code before reporting the defect to the issue tracking system.



technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	**−0.143**	[**−0.250**, **−0.037**]	**0.017**	**47**	**54**
TrpAutoRepair	−0.095	[−0.215, 0.025]	0.134	54	47
AE	−0.113	[−0.225, −0.001]	0.078	49	52
⋃C	−0.075	[−0.201, 0.051]	0.262	59	42
ManyBugs (105-defect subset)					
SPR	−0.120	[−0.366, 0.126]	0.338	25	17
KaliC	0.021	[−0.205, 0.246]	0.882	23	19
Prophet	0.003	[−0.250, 0.255]	1.000	12	30
⋃C	−0.029	[−0.266, 0.209]	0.965	26	16
Defects4J					
GenProgJ	0.022	[−0.208, 0.252]	0.871	18	147
Nopol	0.175	[−0.009, 0.358]	0.096	27	138
KaliJ	−0.032	[−0.264, 0.199]	0.852	14	151
⋃Java	0.105	[−0.071, 0.282]	0.267	35	130

Figure 2.5: Correlation between program versions affected by a defect and repair applicability. Versions data are available for 101 ManyBugs and 165 Defects4J defects. Number of versions affected by a defect likely has little effect on automated repair's effectiveness producing a patch for that defect. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are **bold**. The data shown are only for those defects classified as bugs and with known versions values.

In such cases, our methodology for measuring the time to fix will not capture the time the developer(s) spent thinking about the defect prior to reporting it. Unfortunately,

this time is not recorded in the various surviving artifacts. However, this situation most likely pertains only to defects that the developer(s) fix quickly, thus correctly capturing the rank of the defects' time to fix measure. Top of Figure 2.6 shows the distribution of the time to fix values.

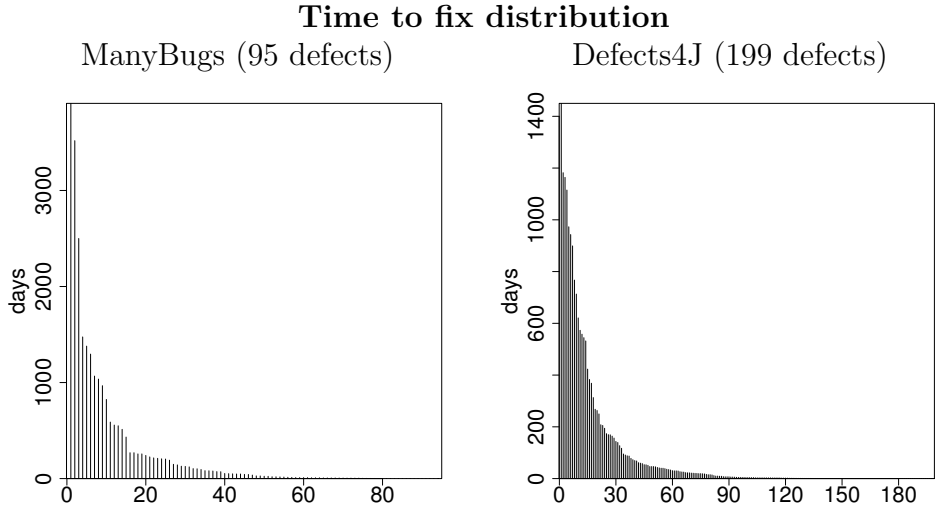
Figure 2.6 shows the results of the Somers' d and the Mann-Whitney U tests on the time taken to fix distributions of defects for which techniques do, and do not produce patches. For every technique, Somers' d indicates a negative correlation: the longer it took for developers to repair a defect, the harder it is for automated repair techniques to produce a patch. However, these results are not statistically significant as indicated by the Mann-Whitney U test, except for GenProgC ($p \leq 0.1$) and GenProgJ ($p \leq 0.05$). For all other techniques, $p > 0.1$. We conclude that the time taken by the developer(s) to fix a defect likely has little effect on automated repair's effectiveness producing a patch for that defect.

These results indicate that Java repair techniques are moderately more likely to patch defects of a higher priority, while C techniques do not correlate with defect priority. The time taken by the developer(s) to fix a defect and number of software versions affected by the defect had little to no correlation with the ability to produce a patch. Overall, there is evidence that automated repair is as likely to repair more important defects, as it is to repair less important ones, which is an encouraging finding.

2.4.2 Defect Complexity

RQ2: Is an APR technique's ability to produce a patch for a defect correlated with that defect's complexity?

We measure a defect's complexity using two parameters, the number of files containing non-comment, non-blank-line edits in the developer-written fix, and the



technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	−0.231	[−0.459, −0.003]	0.053	45	50
TrpAutoRepair	−0.067	[−0.304, 0.170]	0.576	51	44
AE	−0.128	[−0.360, 0.105]	0.287	45	50
⋃C	−0.112	[−0.349, 0.126]	0.357	55	40
ManyBugs (105-defect subset)					
SPR	−0.175	[−0.552, 0.202]	0.382	20	16
KaliC	−0.152	[−0.532, 0.229]	0.447	19	17
Prophet	−0.095	[−0.524, 0.335]	0.688	9	27
⋃C	−0.130	[−0.513, 0.253]	0.521	21	15
Defects4J					
GenProgJ	−0.294	[−0.590, 0.002]	0.024	22	177
Nopol	−0.173	[−0.390, 0.043]	0.137	29	170
KaliJ	−0.131	[−0.381, 0.119]	0.364	18	181
⋃Java	**−0.226**	**[−0.411, −0.040]**	**0.029**	**39**	**160**

Figure 2.6: Correlation between time taken by developer(s) to fix a defect and repair applicability. Time to fix data are available for 95 ManyBugs and 199 Defects4J defects. Time to fix a defect likely has little effect on automated repair's effectiveness producing a patch for that defect. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are **bold**. The data shown are only for those defects classified as bugs and with known time to fix values.

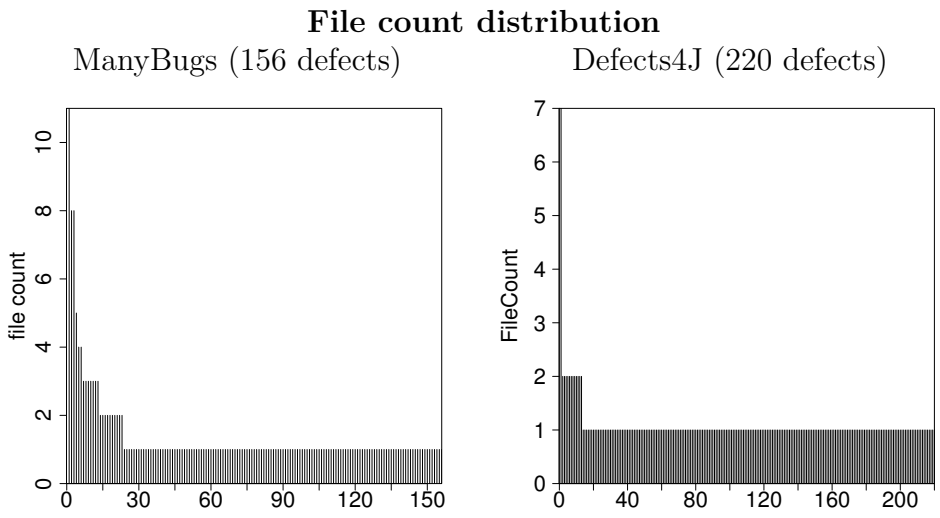
total number of non-comment, non-blank lines of code in the developer-written fix (recall Section 2.2).

Number of source files edited by the developer-written patch. The information on the number of files edited by the developer patch was available for all 185 defects in ManyBugs. The number of files varied from 1 to 11. The distribution of the 156 ManyBugs defects that were classified as bugs also varied from 1 to 11: 133 edited a single file, 10 two files, 7 three files, 2 four files, 1 five files, 2 eight files, and 1 eleven files. For Defects4J too, all 224 defects had this information. The number of files varied from 1 to 7. Of the 220 defects classified as bugs, 205 edited a single file, 12 two files, 1 three files, 1 four files, and 1 seven files. Top of Figure 2.7 shows the distribution of the number of files edited values.

For C techniques, Somers' d showed a weak to moderate negative correlation between the number of files the developer-written patch edited, and the techniques' ability to produce a patch (Figure 2.7). The Mann-Whitney U test showed this relationship to be statistically significant ($p \leq 0.05$) for all C techniques. The correlation was also negative for Java repair techniques, although this relationship was very weak and statistically insignificant ($p > 0.1$). We suspect the relatively weaker correlation for Java programs is due to the lower variability in Defects4J in the number of files edited by the developer patch.

Number of non-blank, non-comment lines of code in developer-written patch. For the 185 ManyBugs defects, the number of non-comment, non-blank lines in the developer-written patches varied from 1 to 1,887. In the subset of ManyBugs consisting of 156 bugs, the number varied from 1 to 1,341. For Defects4J, for both the 224 defects and the subset consisting of 220 bugs, the number of non-comment, non-blank lines in the developer-written patches varied from 1 to 49. Top of Figure 2.8 shows the distribution of the number of lines edited values.

For C techniques, Somers' d showed a weak to strong negative correlation: the larger the developer-written patch, the less likely automated repair is to produce a patch. The Mann-Whitney U test showed this relationship to be significant ($p \leq 0.05$)



technique	Somers' d	95% CI	p	patched	unpatched
		ManyBugs			
GenProgC	**−0.134**	[**−0.241**, **−0.026**]	**0.016**	**75**	**81**
TrpAutoRepair	**−0.117**	[**−0.231**, **−0.002**]	**0.042**	**84**	**72**
AE	**−0.134**	[**−0.241**, **−0.026**]	**0.016**	**75**	**81**
⋃C	**−0.147**	[**−0.268**, **−0.025**]	**0.011**	**91**	**65**
		ManyBugs (105-defect subset)			
SPR	**−0.256**	[**−0.407**, **−0.105**]	**0.002**	**41**	**43**
KaliC	**−0.191**	[**−0.343**, **−0.039**]	**0.028**	**39**	**45**
Prophet	−0.186	[−0.297, −0.075]	0.062	22	62
⋃C	**−0.233**	[**−0.396**, **−0.070**]	**0.007**	**44**	**40**
		Defects4J			
GenProgJ	−0.067	[−0.090, −0.045]	0.232	27	193
Nopol	−0.035	[−0.098, 0.027]	0.521	34	186
KaliJ	−0.066	[−0.085, −0.047]	0.372	22	198
⋃Java	−0.047	[−0.100, 0.005]	0.298	46	174

Figure 2.7: Correlation between number of files modified by developer(s) to fix a defect and repair applicability. Number of files in the developer-written patch data are available for all 156 ManyBugs and 220 Defects4J defects. Automated program repair is less likely to produce patches for defects whose developer-written patches edit more files. This result is strongly statistically significant for C repair techniques, but is statistically insignificant for Java repair techniques. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are **bold**. The data shown are only for those defects classified as bugs.

for all C techniques (see Figure 2.8). For Java techniques, the results were insignificant ($p > 0.1$ or the 95% CI spanned zero). Thus, we cannot conclude that the number of non-comment, non-blank lines in the developer-written patches is significantly associated with repairability for Java techniques.

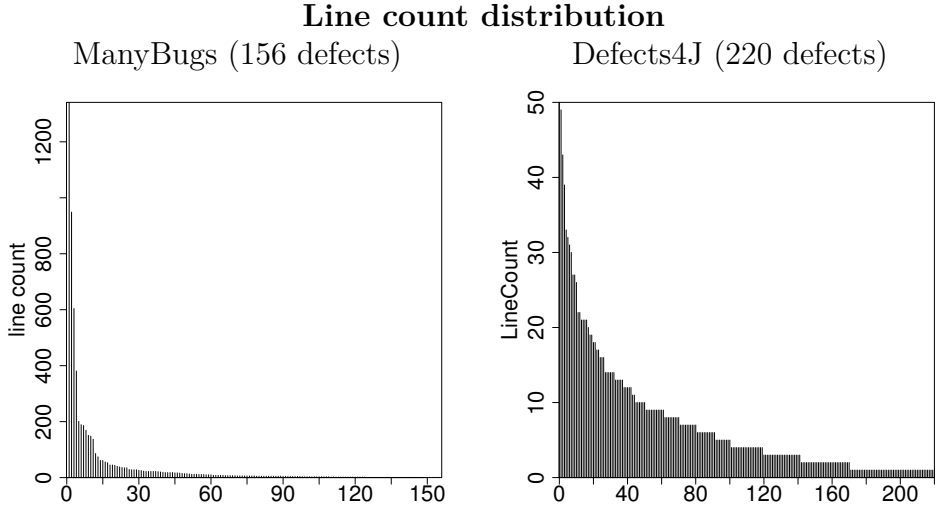
> These results indicate that C repair techniques are less likely to produce patches for defects that required developers to write more lines of code and edit more files to patch. This suggests that automated repair is more likely to patch easy defects than hard ones, reducing its utility. However, the correlation is not strong for all the techniques meaning that automated repair could still produce patches for some hard-to-repair-manually defects.

2.4.3 Test Effectiveness

RQ3: Is an APR technique's ability to produce a patch for a defect correlated with the effectiveness of the test suite used to repair that defect?

We measure a test suite's quality using three parameters, statement coverage, the number of defect-triggering test cases, and the number of relevant test cases (recall Section 2.2).

The fraction of the lines in the files edited by the developer-written patches that are executed by the test suite. For ManyBugs, we were able to compute test suite statement coverage for 113 out of 156 defects classified as bugs. This measure — the fraction of the lines in the files edited by the developer-written patches that are executed by the test suite — varied from 1.6% to 99.4% uniformly across the 113 defects. For Defects4J, we were able to compute test suite statement coverage for all 220 defects classified as bugs. The fraction varied from 7.9% to 100%; for 214 out of 220 defects, the fraction was above 50%, for 5 defects, the fraction was between 30% and 50%, and for 1 defect, the fraction was 7.9%. Top of Figure 2.9 shows the distribution of test suite statement coverage values.



technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	**−0.228**	**[−0.403, −0.054]**	**0.013**	**75**	**81**
TrpAutoRepair	**−0.182**	**[−0.359, −0.005]**	**0.049**	**84**	**72**
AE	**−0.221**	**[−0.396, −0.046]**	**0.016**	**75**	**81**
⋃C	**−0.249**	**[−0.426, −0.072]**	**0.008**	**91**	**65**
ManyBugs (105-defect subset)					
SPR	**−0.405**	**[−0.625, −0.185]**	**0.001**	**41**	**43**
KaliC	**−0.342**	**[−0.569, −0.115]**	**0.006**	**39**	**45**
Prophet	**−0.363**	**[−0.624, −0.102]**	**0.011**	**22**	**62**
⋃C	**−0.388**	**[−0.614, −0.161]**	**0.002**	**44**	**40**
Defects4J					
GenProgJ	−0.083	[−0.318, 0.152]	0.485	27	193
Nopol	0.209	[−0.018, 0.437]	0.050	34	186
KaliJ	−0.072	[−0.315, 0.170]	0.578	22	198
⋃Java	0.060	[−0.128, 0.247]	0.533	46	174

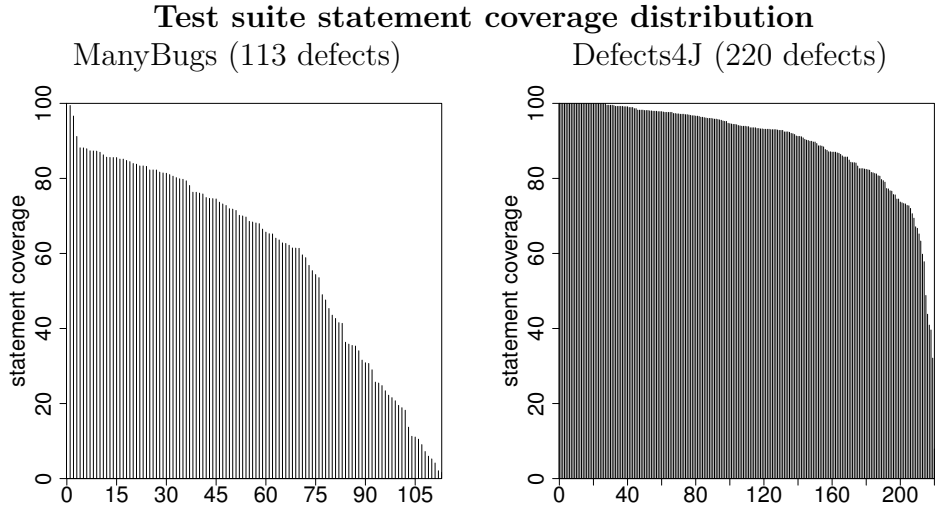
Figure 2.8: Correlation between number of non-comment, non-blank lines of files modified by developer(s) to fix a defect and repair applicability. The number of non-comment, non-blank lines of files in the developer-written patch data are available for all 185 ManyBugs and 220 Defects4J defects classified as bugs. This number is strongly correlated with automated repair techniques' ability to produce patches. This result is strongly statistically significant for C repair techniques and Nopol. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are **bold**. The data shown are only for those defects classified as bugs.

For C and Java techniques, the results were insignificant. Somers' d showed a very weak to weak correlation between the coverage of the test suite used to repair the defect and the automated repair's ability to produce a patch for that defect (Figure 2.9); the 95% CI for Somers' d consistently spanned zero and the Mann-Whitney U test showed that the differences between the distributions are not statistically significant ($p > 0.1$ for all techniques).

The number of defect-triggering test cases. For ManyBugs, all 156 defects classified as bugs had information on the number of test cases that trigger the defect. This number of tests varied from 1 to 52. Of these 156 defects, 111 had only a single triggering test case. For Defects4J, all 220 defects had this information, varying from 1 to 28. Of the 220 defects, 152 had only a single triggering test. Top of Figure 2.10 shows the distribution of triggering test counts.

For C techniques, the results were insignificant ($p > 0.1$ or the 95% CI spanned zero). For Java techniques, Somers' d showed a weak to moderate negative correlation between the number of triggering test cases and the ability to produce a patch (Figure 2.10). The Mann-Whitney U test indicated this result to be statistically significant ($p \leq 0.05$) for all Java techniques except KaliJ, for which $p \leq 0.1$. We conclude that the number of triggering tests negatively affects a Java technique's ability to produce a patch.

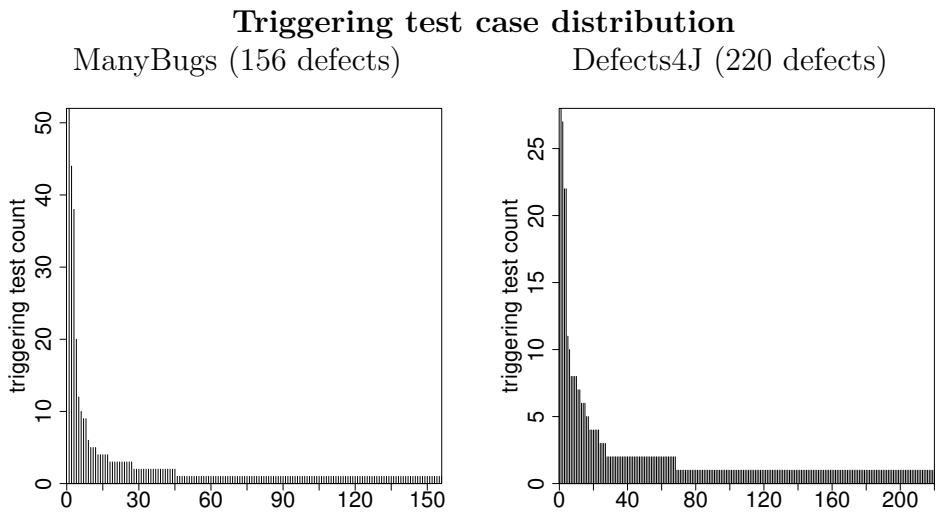
The number of relevant test cases (test cases that execute at least one line of the developer-written patch). For ManyBugs, we annotated the same 156 defects with the total number of positive and negative test cases provided for each defect in ManyBugs benchmark. The number of relevant test cases varied from 3 to 7,951. For Defects4J, we annotated the 220 defects with the number of relevant tests provided for each defect in Defects4J benchmark and the number of relevant test cases varied from 1 to 4011. Top of Figure 2.11 shows the distribution of relevant test counts.



technique	Somers' d	95% CI		p	patched	unpatched
ManyBugs						
GenProgC	−0.094	[−0.312,	0.124]	0.390	55	58
TrpAutoRepair	−0.088	[−0.304,	0.127]	0.421	60	53
AE	−0.159	[−0.374,	0.057]	0.146	57	56
⋃C	−0.166	[−0.378,	0.046]	0.134	66	47
ManyBugs (105-defect subset)						
SPR	−0.073	[−0.365,	0.219]	0.640	34	25
KaliC	−0.090	[−0.383,	0.203]	0.558	32	27
Prophet	−0.109	[−0.484,	0.266]	0.538	15	44
⋃C	−0.094	[−0.387,	0.198]	0.550	36	23
Defects4J						
GenProgJ	−0.097	[−0.354,	0.160]	0.416	27	193
Nopol	−0.064	[−0.278,	0.149]	0.555	34	186
KaliJ	−0.158	[−0.468,	0.152]	0.226	22	198
⋃Java	−0.011	[−0.204,	0.181]	0.907	46	174

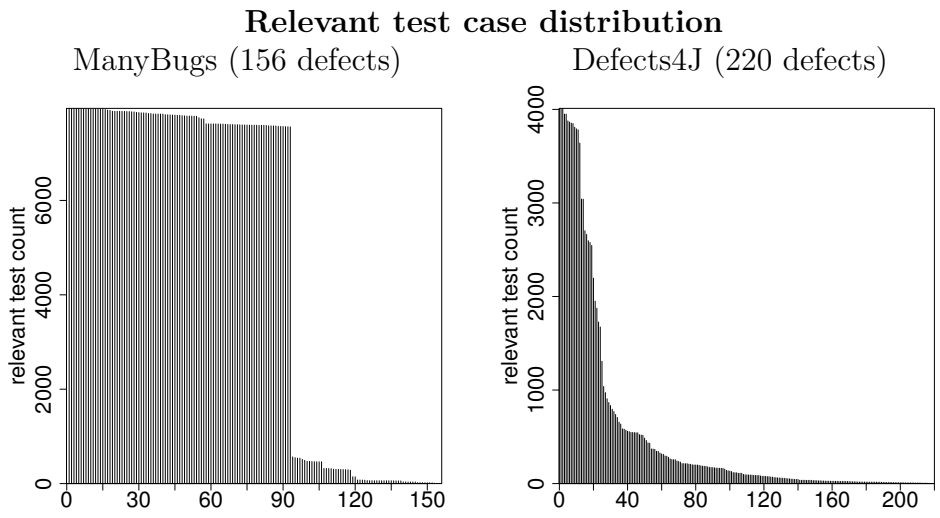
Figure 2.9: Correlation between the statement coverage of developer tests and repair applicability. The statement coverage of the test suite are available for 113 ManyBugs and 220 Defects4J defects. There is a weak, statistically insignificant, negative correlation for all techniques, between the coverage of the test suite used to repair the defect, and the technique's ability to produce a patch for the defect. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. The data shown are only for those defects classified as bugs for which we could compute coverage information.

For C techniques, Somers' d showed a very weak positive correlation for TrpAutoRepair. The Mann-Whitney U test indicated this result to be statistically



technique	Somers' d	95% CI	p	patched	unpatched
ManyBugs					
GenProgC	0.098	[−0.047, 0.243]	0.187	75	81
TrpAutoRepair	0.137	[−0.004, 0.278]	0.065	84	72
AE	0.067	[−0.079, 0.212]	0.370	75	81
⋃C	0.087	[−0.056, 0.229]	0.249	91	65
ManyBugs (105-defect subset)					
SPR	0.018	[−0.182, 0.217]	0.866	41	43
KaliC	−0.013	[−0.212, 0.186]	0.905	39	45
Prophet	−0.063	[−0.266, 0.140]	0.589	22	62
⋃C	−0.025	[−0.225, 0.175]	0.810	44	40
Defects4J					
GenProgJ	**−0.231**	[**−0.348**, **−0.113**]	**0.017**	**27**	**193**
Nopol	**−0.221**	[**−0.337**, **−0.104**]	**0.012**	**34**	**186**
KaliJ	−0.177	[−0.342, −0.013]	0.095	22	198
⋃Java	**−0.252**	[**−0.348**, **−0.156**]	**0.001**	**46**	**174**

Figure 2.10: Correlation between the number of failing tests that trigger a defect and repair applicability. The number of triggering test cases is available for all 156 ManyBugs and 220 Defects4J defects. There is a negative correlation between a defect's number of triggering test cases and the ability to produce a patch, but this relationship is only statistically significant for the Java repair techniques. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are **bold**. The data shown are only for those defects classified as bugs.



technique	Somers' d	95% CI		p	patched	unpatched
ManyBugs						
GenProgC	0.166	[−0.017,	0.349]	0.074	75	81
TrpAutoRepair	**0.188**	[**0.008**,	**0.368**]	**0.043**	**84**	**72**
AE	0.122	[−0.062,	0.307]	0.188	75	81
⋃C	0.134	[−0.048,	0.316]	0.154	91	65
ManyBugs (105-defect subset)						
SPR	0.087	[−0.162,	0.335]	0.497	41	43
KaliC	0.085	[−0.164,	0.334]	0.507	39	45
Prophet	−0.096	[−0.396,	0.204]	0.509	22	62
⋃C	0.050	[−0.199,	0.299]	0.697	44	40
Defects4J						
GenProgJ	−0.204	[−0.432,	0.024]	0.086	27	193
Nopol	**−0.317**	[**−0.549**,	**−0.084**]	**0.003**	**34**	**186**
KaliJ	−0.223	[−0.505,	0.059]	0.087	22	198
⋃Java	**−0.313**	[**−0.500**,	**−0.125**]	**0.001**	**46**	**174**

Figure 2.11: Correlation between the number of tests relevant to a defect and repair applicability. The number of relevant test cases is available for all 156 ManyBugs and 220 Defects4J defects. For Java repair techniques, there is a weak to moderate significant negative correlation between a defect's number of relevant test cases and the ability to produce a patch. For C repair techniques, the correlation is weakly positive. These correlations are statistically significant for a subset of techniques. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are **bold**.

significant ($p \leq 0.05$). However, our confounding factor analysis (Section 2.5.3) found that this correlation was due to a correlation between the number of relevant test cases and the number of files edited by the developer-written patch. For all other C techniques, the results were insignificant ($p > 0.1$ or the 95% CI spanned zero). For Java techniques, Somers' d showed a moderate to strong negative correlation between the number of relevant test cases and the ability to produce a patch for all techniques (Figure 2.11). The correlation was statistically significant ($p \leq 0.05$) for Nopol and for $\bigcup$Java and weakly statistically significant ($p \leq 0.1$) for GenProgJ and KaliJ.

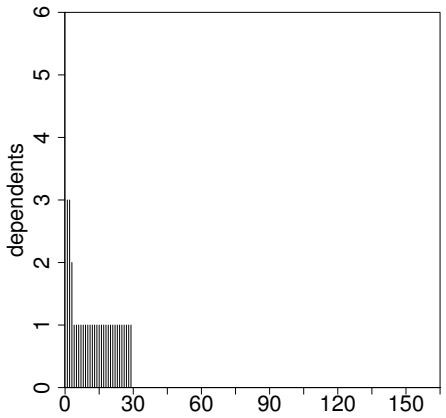
> These results indicate that Java repair techniques are less likely to produce patches for defects with more triggering or more relevant tests. Test suite coverage does not significantly correlate with the ability to produce a patch. These findings are concerning, as they show it is harder to produce patches in situations that prior work has shown to lead to higher-quality patches [252].

2.4.4 Defect Independence

RQ4: Is an APR technique's ability to produce a patch for a defect correlated with that defect's dependence on other defects?

Our dataset turned out to be insufficient to draw conclusions on a relationship between independence and repairability. For ManyBugs, 76 out of 156 defects classified as bugs had information on how many other defects they depended on, but none of them depended on other defects. For Defects4J, 165 out of 220 defects classified as bugs had this information. Of these, 136 did not depend on other defects, 26 depended on a one other defect, 1 on two other defects, and 2 on three other defects.

For C techniques, the lack of variability in the benchmark defects with respect to defect independence prevented us from drawing any conclusions. For Java techniques, the results were insignificant ($p > 0.1$ or the 95% CI spanned zero), as Figure 2.12



	Defects4J					
technique	Somers' d	95% CI		p	patched	unpatched
GenProgJ	−0.075	[−0.233,	0.083]	0.494	18	147
Nopol	0.058	[−0.114,	0.230]	0.501	27	138
KaliJ	−0.039	[−0.230,	0.153]	0.767	14	151
⋃Java	0.032	[−0.117,	0.181]	0.725	35	130

Figure 2.12: Correlation between the a defect's dependence on other defects and repair applicability. For Java repair techniques, there is a weak statistically insignificant correlation between a defect's dependence on other defects and the ability to produce a patch. The 95% CI (confidence interval) column shows the range in which Somers d lies with a 95% confidence. The data shown are only for those defects classified as bugs.

shows. This suggests that the number of other defects a defect depends on does not affect repairability of Java repair techniques.

While we have developed a methodology that can be applied to other defect benchmarks, ManyBugs did not contain enough variability in defect independence to identify a relationship between independence and repairability. For Defects4J, an insignificant correlation is observed for all the techniques.

2.4.5 Developer-Written Patch Characteristics

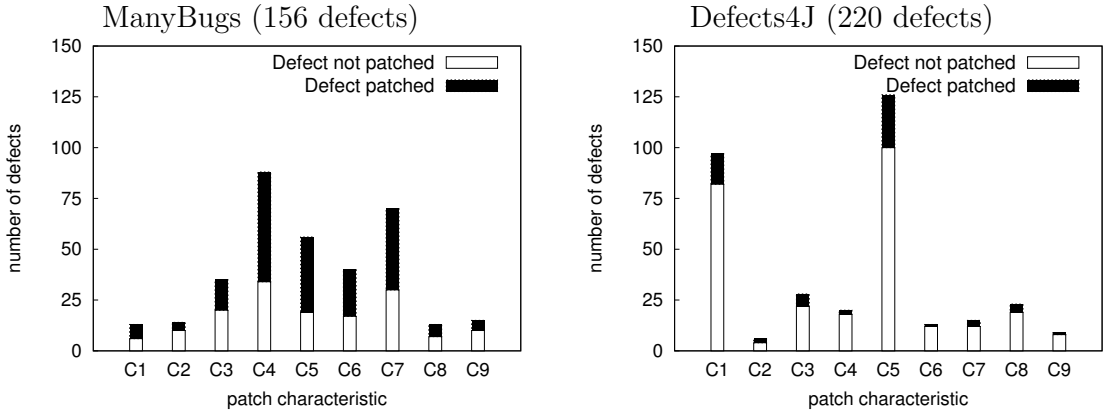
RQ5: What characteristics of the developer-written patch are significantly associated with an APR technique's ability to produce a patch for that defect?

Investigating which characteristics of the developer-written patches are significantly associated with defect repairability allows us to reason about automated repair's ability to fix defects in terms of what the developers did. This may, in turn, lead to actionable advice about which kinds of defects the developer(s) should fix manually, and which can be trusted to automated repair. Of course, to use this information, the developer(s) must have a sense of the characteristics of the patch before it is written, which may sometimes be possible. However, the main goal of studying this research question is to help guide future research into automated program repair techniques by identifying the characteristics of the defects, in terms of the patches that repair them, that existing techniques struggle to produce patches for. Research into future repair tools may, for example, target modifying or inserting loops, just as, for example, Nopol targets conditional statements.

Figure 2.13 shows the distributions of the nine patch characteristics for the two benchmarks, and the results of a logistic regression using these characteristics. For each repair technique, Figure 2.13 shows which patch characteristics are significantly associated with repairability and how much variance in repairability is explained by all defect characteristics.

The data suggest that some characteristics of developer-written patches are significantly associated with repairability for C repair techniques, but not for Java repair techniques. In particular, for C repair techniques, changing a data structure or type, a function argument, or a conditional, or adding a new variable, an if statement, or a new function are significantly associated with repairability, whereas changing a method signature, or adding a function call, or a loop is not.

Distributions of the nine characteristics of the developer-written patches



technique	model quality p	model quality R^2	patch characteristic characteristic #	patch characteristic p
GenProgC	**0.001**	**0.129**	**C3**	**0.036**
			C4	0.088
			C5	0.066
			C7	0.055
			C9	0.051
TrpAutoRepair	**0.005**	**0.109**	C3	0.058
			C6	**0.046**
			C7	**0.040**
			C9	**0.049**
AE	**0.042**	**0.081**	C2	0.057
			C7	0.068
SPR	**0.001**	**0.184**	C1	0.059
			C3	**0.014**
Prophet	**0.004**	**0.169**	**C3**	**0.003**
KaliC	0.213	0.100	C3	0.088
⋃C	**0.000**	**0.162**	**C3**	**0.001**
			C7	0.075
			C9	**0.030**
GenProgJ	0.555	0.047	C1	0.057
			C5	**0.029**
Nopol	0.572	0.040	none	
KaliJ	0.543	0.055	**C1**	**0.031**
⋃Java	0.395	0.041	**C1**	**0.039**

Figure 2.13: Association between patch characteristics and repair applicability. Top charts show the distribution of the nine patch characteristics for the developer-written patches in the ManyBugs and Defects4J benchmarks. Bottom shows logistic regression reporting that characteristics C1, C3, C5, C6, C7 and C9 — changing a data structure or type, a function argument, or a conditional, or adding a new variable, an if statement or a new function — are significantly associated with repairability. Data for which the p value is below 0.05 are **bold**.

These results suggest that defects that required developers to insert a loop or a new function call, or change a method signature are challenging for automated repair techniques to patch. More patch characteristics are significantly associated with repairability for C repair techniques than for Java repair techniques.

2.4.6 Patch Quality

RQ6: What characteristics of defect are significantly associated with an APR technique's ability to produce a high-quality patch for that defect?

Recent work has begun evaluating the quality of patches produced by automated repair [29, 67, 175, 177, 184, 216, 227, 252]. Until now, our analysis remained quality agnostic, focusing on whether techniques can produce patches, as opposed to whether techniques can produce high-quality patches. Quality and applicability are orthogonal aspects of program repair: one can work on improving the quality of the produced patches, the applicability of the repair techniques to a wider range of defects, or both. However, it is important to also study how the two interact. At the present time, the quality of the patches produced by most techniques is fairly low. According to a manual analysis, on the 105-defect subset of ManyBugs, GenProgC could only produce 2 correct patches, TrpAutoRepair 3 correct patches, and AE 2 correct patches [227]. On an 84-defect subset of Defects4J, GenProgJ could only produce 5 correct patches, Nopol 5 correct patches, and KaliJ 1 correct patch [184]. The number of correct patches is too small for us to make statistically significant claims for these techniques. Inspired by the findings of the low quality of repairs, SPR and Prophet were designed to specifically improve repair quality. SPR produces 13 and Prophet 15 correct patches on the 105-defect subset of ManyBugs [177]. Other techniques that claim to produce high-quality patches, e.g., SearchRepair [122], fail to scale to the size and complexity of real-world defects we consider. We use the SPR and Prophet data to begin studying the defect characteristics' effect on the ability to produce high-quality repairs. There

are eight abstract parameters that have sufficient data to perform such analysis. Figure 2.14 shows the Somers' d and Mann-whitney U test results testing for an association between each of these eight abstract parameters and the ability to produce high-quality patches. Only the abstract parameters related to defect complexity and test suite effectiveness exhibit statistically significant associations.

Defect complexity. For Prophet, the number of non-comment, non-blank lines in the developer-written patch correlated negatively with the ability to produce a correct patch. This negative correlation was stronger compared to the negative correlation with the ability to produce a patch at all ($d = -0.564$ for correct patches, vs. $d = -0.342$ for all patches). For SPR, the correlations with the ability to produce a correct patch and a patch all were the same ($d = -0.405$). The Mann-Whitney U test confirmed this distribution difference to be statistically significant ($p \leq 0.05$) for both the techniques. However, for SPR, the 95% CI for Somers' d spans zero for producing *correct* patches.

For Prophet, there was a weakly significant ($p \leq 0.1$) negative correlation between the number of files edited by the developer-written patch and the ability to produce a correct patch. For SPR, the correlation was insignificant.

Test suite effectiveness. SPR showed a weakly significant ($p < 0.1$) positive correlation for producing correct patches when using higher-coverage test suites ($d = 0.312$ for correct patches, vs. $d = -0.073$ for all patches). This is consistent with prior results showing that higher-coverage test suites lead to higher-quality patches [252]. The result for Prophet was not statistically significant ($p > 0.1$). Also, correlations with the number of triggering tests and relevant tests were either the same for the correct patches as all patches, or not statistically significant.

Developer-written patch characteristics. A logistic regression using the nine patch characteristics showed that characteristics C1 and C3 associated with SPR's ability to produce patches, and characteristic C3 associated with Prophet's ability to

abstract parameter	technique	ManyBugs (105-defect subset) Somers' d	95% CI	p	patched	unpatched
line count	**SPR (*produces patch*)**	**−0.405**	[**−0.655**, **−0.155**]	**0.001**	**41**	**43**
	SPR (*correct patch*)	−0.405	[−0.826, 0.016]	0.023	12	72
	Prophet (*produces patch*)	**−0.342**	[**−0.594**, **−0.090**]	**0.006**	**39**	**45**
	Prophet (*correct patch*)	**−0.564**	[**−0.963**, **−0.165**]	**0.001**	**14**	**70**
file count	**SPR (*produces patch*)**	**−0.256**	[**−0.402**, **−0.111**]	**0.002**	**41**	**43**
	SPR (*correct patch*)	−0.120	[−0.264, 0.024]	0.328	12	72
	Prophet (*produces patch*)	**−0.191**	[**−0.339**, **−0.042**]	**0.028**	**39**	**45**
	Prophet (*correct patch*)	−0.214	[−0.284, −0.144]	0.093	14	70
statement coverage	SPR (*produces patch*)	−0.073	[−0.366, 0.220]	0.640	34	25
	SPR (*correct patch*)	0.312	[−0.071, 0.695]	0.099	12	47
	Prophet (*produces patch*)	−0.090	[−0.385, 0.204]	0.558	32	27
	Prophet (*correct patch*)	0.284	[−0.101, 0.668]	0.135	12	47
triggering test count	SPR (*produces patch*)	0.018	[−0.182, 0.217]	0.866	41	43
	SPR (*correct patch*)	0.109	[−0.208, 0.426]	0.470	12	72
	Prophet (*produces patch*)	−0.013	[−0.212, 0.186]	0.905	39	45
	Prophet (*correct patch*)	0.118	[−0.179, 0.416]	0.390	14	70
relevant test count	SPR (*produces patch*)	0.087	[−0.159, 0.332]	0.497	41	43
	SPR (*correct patch*)	0.319	[−0.100, 0.739]	0.078	12	72
	Prophet (*produces patch*)	0.085	[−0.162, 0.332]	0.507	39	45
	Prophet (*correct patch*)	0.272	[−0.093, 0.638]	0.110	14	70
	SPR (*produces patch*)	−0.111	[−0.510, 0.288]	0.753	9	5
	SPR (*correct patch*)	−0.154	[−0.366, 0.058]	1.000	1	13
	Prophet (*produces patch*)	−0.111	[−0.510, 0.288]	0.753	9	5
	Prophet (*correct patch*)	NA	NA	NA	0	14
versions	SPR (*produces patch*)	−0.120	[−0.358, 0.118]	0.338	25	17
	SPR (*correct patch*)	−0.206	[−0.307, −0.104]	0.312	8	34
	Prophet (*produces patch*)	0.021	[−0.205, 0.246]	0.882	23	19
	Prophet (*correct patch*)	−0.206	[−0.307, −0.104]	0.312	8	34
time to fix	SPR (*produces patch*)	−0.175	[−0.555, 0.205]	0.382	20	16
	SPR (*correct patch*)	−0.172	[−0.681, 0.337]	0.501	7	29
	Prophet (*produces patch*)	−0.152	[−0.537, 0.233]	0.447	19	17
	Prophet (*correct patch*)	−0.192	[−0.710, 0.326]	0.452	7	29

Figure 2.14: Correlation between defect characteristics and repair quality. Defect complexity and test suite effectiveness exhibit statistically significant associations with the techniques' ability to produce high-quality patches. SPR and Prophet are the only two techniques that produce a sufficient number of high-quality patches for this analysis. The 95% CI (confidence interval) column shows the range in which Somers' d lies with a 95% confidence. Rows for which both the Mann-Whitney U test produces a p value below 0.05 and the 95% CI does not span zero are **bold**. The data shown are only for those defects classified as bugs and with known respective parameter values.

produce patches (recall Figure 2.13). A logistic regression of *correct* patches generated by SPR and Prophet identified the same characteristics associating with producing correct patches, and also identified C7 (patch adds an if statement) as statistically significantly associating with producing correct patches (SPR $p = 0.044$, Prophet $p = 0.086$). Both SPR and Prophet target defects that can be repaired by inserting or

modifying conditional statements, explaining the observation that adding if statements associates with producing correct patches.

None of the statistical tests revealed statistically significant results for the correct patches for the defect importance and defect independence characteristics.

> Only two of the considered repair techniques, Prophet and SPR, produce a sufficient number of high-quality patches to evaluate. These techniques were less likely to produce patches for more complex defects, and they were even less likely to produce correct patches.

2.4.7 Feature Synthesis

We wanted to conduct the same statistical tests to measure correlation between defect characteristics with the ability to synthesize features using those defects in our benchmarks that are features, not bugs. Unfortunately, too few of the defects were features: Features make up 29 of the 185 defects in ManyBugs (21 of the 105-defect subset), and only 4 of the 224 in Defects4J. GenProgJ and KaliJ synthesize none of the features and Nopol synthesizes 1. Meanwhile GenProgC synthesized 11, TrpAutoRepair synthesized 12 and AE synthesized 10 out of 29 features from 185-subset of ManyBugs and SPR synthesized 5, Prophet synthesized 4 and KaliC synthesized 5 out of 21 features from 105-subset of ManyBugs. These sample sizes are too small and none of our experiments revealed statistically significant results.

2.5 Discussion

This section discusses the implications of our findings (Section 2.5.1), makes observations about our dataset and the use of the methodology that produced it (Section 2.5.2), and analyzes potential confounding factors within our evaluation (Section 2.5.3).

2.5.1 Implications

Our data suggest several encouraging conclusions. First, APR techniques are slightly more likely to produce patches for defects of a higher priority, and are equally likely to produce patches for defects regardless of how long developer(s) took to fix them. The former finding may suggest differences between low- and high-priority defects, from the point of view of automated program repair. Second, while overall, automated repair techniques were more likely to produce patches for defects that required fewer edits by the developers to fix, the correlations were not strong for all techniques, and the techniques were able to produce patches for some hard-to-repair-manually defects. Producing larger patches requires search-based automated repair techniques to explore more of the search space, which requires longer execution time. As repair techniques typically operate with a time limit, finding such patches may be more difficult than smaller ones. The fact that techniques were able to find patches for some defects that were hard to repair manually suggests that either the techniques are able to sometimes successfully traverse the large search space, or that smaller patches exist than the manually written ones.

At the same time, our data suggest that Java repair techniques had a harder time producing patches for defects with more triggering or more relevant tests. This finding is intuitive because each test executing code related to the defect represents constraints on the patch. To produce a patch, the automated repair techniques have to modify the code to satisfy all the constraints. The more constraints there are, the harder it is to find a satisfying patch. Prior studies have found that higher-coverage test suites can lead to higher-quality patches [252] and that larger search spaces lead to a higher fraction of incorrect patches [176]. As a result, we find that test suites that make it easier to produce a patch reduce, in expectation, the quality of the produced patch. This identifies a research challenge of creating techniques that are capable of either finding patches effectively even when constrained by high-quality test suites, or

discriminating between low-quality and high-quality patches despite using test suites that provide few guiding constraints.

We identified some evidence that targeting repair techniques to specific defects is worthwhile, as defects that required a developer to write new if statements were more likely to be correctly repaired by SPR and Prophet, two techniques designed to insert or modify conditional statements. This provides preliminary evidence that perhaps when automated repair techniques are applied to defects that developers patched using the kinds of changes the techniques are designed to make, the techniques are capable of making higher-quality changes.

2.5.2 Dataset Observations

The ManyBugs and Defects4J datasets lack certain kinds of data diversity to answer some of our proposed research questions. For example, every defect in ManyBugs that included dependence information did not depend on other defects. Similarly, only two of the evaluated repair techniques produced sufficiently many high-quality patches for our analysis to make statistically significant findings about patch quality. Nevertheless, this study presents a methodology that can be applied to other datasets to derive more data to answer these questions, particularly as the body of defects on which automated program repair techniques are evaluated grows.

One of the goals of our study has been to create a methodology for evaluating the applicability of automated program repair techniques that can be applied to new techniques and help drive research toward improving such applicability. As such, none of the defect characteristics we consider are specific to a repair technique. For example, we define defect complexity in terms of the number of lines and number of files edited by the minimized developer-written patch, and how easy it is to reproduce the defect. We do not take into account that some repair techniques may, for example, find defects that involve control flow more complex than ones that do not. Our study of RQ5

empirically identifies several characteristics of the defects' developer-written patches (such as if the patch changes a conditional or adds a function argument) that associate with the techniques' ability to produce patches for those defects. Studying technique-specific complexity of defects is also a worthwhile effort, but it is beyond the scope of our work on creating a technique-agnostic applicability evaluation methodology.

2.5.3 Confounding Factor Analysis

To consider potential confounding factors in our analyses, we computed the Spearman correlation coefficients between all pairs of abstract parameters. Figure 2.15 shows these coefficients for ManyBugs and Defects4J. The **bold** coefficients are statistically significant ($p \leq 0.05$) and underlined coefficients are weakly statistically significant ($p \leq 0.1$). To be conservative in our analysis, we consider all pairs that correlate at least weakly significantly ($p < 0.1$) to pose potential confounding factors. We found that for ManyBugs, the following pairs of cross-defect-characteristic parameters correlated at least weakly significantly: file count correlates with relevant test count and triggering test count, line count correlates with triggering test count and statement coverage, and versions correlates with relevant test count. For Defects4J, relevant test count correlates with time to fix and priority. (All other correlations of at least weak statistical significance were within defect characteristics, e.g., file count correlated with line count.)

For each correlating cross-characteristic parameter pair $\langle p_1, p_2 \rangle$, we created four logistic regression models for repairability:

Model_1: a model using only p_1,

Model_2: a model using only p_2,

Model_{1+2}: a model using a linear combination of p_1 and p_2 ($p_1 + p_2$), and

Model_{1*2}: a model using all possible interactions between p_1 and p_2 ($p_1 * p_2$).

We then pairwise compare the models' goodness of fit using the area under the curve

ManyBugs								
	file count	line count	relevant test count	triggering test count	statement coverage	time to fix	versions	priority
line count	**0.46**							
relevant test count	**−0.22**	−0.13						
triggering test count	**0.05**	**0.14**	−0.04					
statement coverage	−0.01	−<u>0.14</u>	**0.30**	0.03				
time to fix	0.07	0.10	−0.19	0.11	0.02			
versions	0.11	0.10	**−0.30**	−0.16	−0.12	0.18		
priority	−0.23	−0.50	0.12	−0.14	0.26	0.12	0.06	
dependents	—	—	—	—	—	—	—	—

Defects4J								
	file count	line count	relevant test count	triggering test count	statement coverage	time to fix	versions	priority
line count	0.17							
relevant test count	0.17	0.08						
triggering test count	0.11	0.04	0.22					
statement coverage	0.00	−0.02	−0.01	<u>−0.03</u>				
time to fix	0.05	0.13	**0.22**	0.11	−0.02			
versions	−0.10	0.02	−0.02	−0.02	0.00	−0.15		
priority	0.03	0.11	**−0.13**	−0.05	0.04	−0.19	**0.10**	
dependents	0.04	0.09	−0.05	−0.10	0.05	0.17	−0.11	0.09

Figure 2.15: Confounding parameter analysis in annotated defect benchmarks. Pairwise Spearman correlation coefficients for the abstract parameters for the ManyBugs and Defects4J defects. The **bold** coefficients are statistically significant ($p \leq 0.05$) and <u>underlined</u> coefficients are weakly statistically significant ($p \leq 0.1$).

and determine the statistical significance in the models' quality improvement. We consider improvements that are at least weakly statistically significant to demonstrate confounding factors. If Model_{1+2} shows a significant improvement over model Model_1, we determine that parameter p_2's contribution to the model is significant. Similarly, if Model_{1+2} improves over Model_2 then p_1 contributes. Finally, if Model_{1*2} improves significantly on Model_{1+2}, then there exists an interaction between p_1 and p_2 whose contribution is significant.

Analyzing the correlated pairs of parameters, we find that:

- For all C techniques except TrpAutoRepair, relevant test count does not contribute significantly to model quality beyond file count's contribution. We conclude, for our C analysis, that the observed correlation (weak, significant for TrpAutoRepair) between relevant test count and repairability (Section 2.4.3)

is not due to the confounding factor of relevant test count correlating with file count. The interactions between relevant test count and file count do not significantly contribute to the quality of model.

- For all C techniques, triggering test count does not contribute significantly to model quality beyond file count's contribution; however, for GenProgC, the interactions of file count and triggering test count do offer a significant contribution. We conclude, for our C analysis, that the observed correlation (moderate, weakly significant for TrpAutoRepair) between triggering test count and repairability (Section 2.4.3) is likely largely due to the confounding factor of triggering test count correlating with file count, and file count correlating with repairability, although the combination of the two parameters does add some useful information.

- For GenProgC, TrpAutoRepair, SPR, Prophet, and $\bigcup$C on the full 185-defect ManyBugs, both line count and triggering test count contribute significantly to model quality. For TrpAutoRepair, Prophet, and $\bigcup$C, ManyBugs, interactions between the two parameters significantly contribute more than the individual contributions of the parameters. We conclude, for our C analysis, that the correlation between line count and triggering test count is not a confounding factor.

- For each C technique except AE and $\bigcup$C on 185-defect ManyBugs, both line count and statement coverage contribute significantly to model quality. The interactions between the two offer no significant contribution. We conclude, for our C analysis, that the correlation between line count and statement coverage is not a confounding factor.

- For each C technique except GenProgC, both relevant test count and versions contribute significantly to model quality. The interactions between the two

parameters do not contribute significantly more than the two parameters on their own. We conclude, for our C analysis, that the correlation between relevant test count and versions is not a confounding factor.

- For each Java technique except Nopol, both relevant test count and time to fix contribute significantly to model quality. For Nopol, time to fix does not contribute significantly to model quality beyond relevant test count's contribution. The interactions between the two parameters do not contribute additional information. We conclude, for our Java analysis, that the correlation between relevant test count and time to fix is not a confounding factor.

- For each Java technique except GenProgJ and KaliJ, priority and relevant test count contribute significantly to the model quality. For GenProgJ and KaliJ, priority does not contribute significantly beyond relevant test count's contribution. The interactions between the two parameters do not contribute significantly more than the two parameters on their own. We conclude, for our Java analysis, that the observed correlations (moderate, weakly significant for GenProgJ and KaliJ) between priority and repairability (Section 2.4.1) is likely largely due to the confounding factor of priority correlating with relevant test count, and relevant test count correlating with repairability.

We conclude that the number of files edited by the developer-written patch is a confounding factor to relevant and triggering test count in the ManyBugs dataset, and that relevant test count is a confounding factor to priority in the Defects4J dataset. All other observed correlations between the parameters do not indicate confounding factors. Our earlier conclusions are not affected by this analysis as only weak or weakly significant observed correlations in only a few cases are affected by the confounding factors, and those observed correlations did not lead to conclusions.

2.6 Threats to Validity

This study investigates the relationship between automated repair techniques' ability to produce patches and characteristics of the defects, test suites, and developer-written patches for the defects. However, this study only begins to explore the relationship between these characteristics and the *quality* of the patches (recall RQ6). Future work needs to address this concern, as today, techniques repair very few of the studied defects correctly, reducing the power of our analysis. The methodology presented in this study can be applied to other defect benchmarks as they become available, and to other repair techniques that focus on repair quality and applicability.

The goal of our study is to characterize the kinds of defects for which automated program repair is capable of producing patches and high-quality patches. The study is observational. Of course, given a defect, changing its metadata, such as its priority, will not affect the techniques' ability to produce patches, and our findings should be viewed as directing research into improving or creating new automated program repair techniques, not as methods for making existing repair techniques apply to specific defects. However, some of our findings suggest how altering inputs to automated program repair techniques may affect patch production, such as that increasing the number of tests may make it more difficult to produce a patch.

We took steps to ensure that our study is objective and reproducible. All characteristics derived from source-code repositories and developer-written patches are computed using deterministic scripts, available at:
`https://github.com/LASER-UMASS/AutomatedRepairApplicabilityData/`.
However, some of the judgments with respect to which parameters are relevant to this study are subjective. We addressed this threat by having two authors independently collect parameters in eleven issue tracking systems, and independently measure all subjective parameters for the defects we considered. The authors then merged their findings, paying special attention to any disagreements in initial judgments.

Our study relies on repairability and quality results presented by prior work [149, 150, 175, 177, 184, 227] and errors or subjective judgment quality in that work affects our findings. As an example, GenProgC makes an assumption that the source code files that must be edited to repair a defect are known [149, 150]; this assumption may not hold in practice, and that would threaten the generalizability of those repairability results, and, in turn, our findings. Using results from multiple studies partially mitigates this threat, although using an objective measure of quality, such as number of independent tests a patch passes [252], would go farther. Unfortunately, such an evaluation requires multiple, independent, high-quality test suites for real-world defects, and such test suites do not exist for the ManyBugs benchmark, but do for Defects4J.

While GenProgC was designed prior to ManyBugs, the other techniques have been developed in part to compete with GenProgC on the then known (at least partially) ManyBugs benchmark. This may affect the generalizability of the techniques to other defects, which, in turn affects the generalizability of our results. We mitigate this threat by using two defect benchmarks and multiple repair techniques.

Our study treats all parameters related to a characteristic as equally important. This is likely an oversimplification of the real world. For example, defect priority is likely a better indicator of a defect's importance than the number of project versions the defect affects. To mitigate this threat, we perform independent analyses with each parameter.

Our study treats all defects related to the same issue in an issue tracking system equally. This may contribute to noisy data. For example, a single issue, and commit, may resolve two defects. One of these defects may be critical, while the other is not. Our analysis, due to lack of finer granularity in the source code repository and issue tracking system, considers both defects as critical, potentially affecting our findings.

The relatively low number of such defects mitigates this threat, although another study could remove such defects altogether.

We consider a developer-written patch to be a proxy of the complexity of the defect it patches. In theory, there may be many other patches for the same defect, some smaller and simpler and others larger and more complex. We mitigate this threat by considering a large number of defects and using benchmarks of mature software projects that are less likely to accept poorly written patches.

2.7 Contributions

Automated program repair has recently become a popular area of research, but most evaluations of repair techniques focus on how many defects a technique can produce a patch for. This study, for the first time, analyzes how characteristics of the defects, the test suites, and the developer-written patches correlate with the repair techniques' ability to produce patches, and to a smaller degree, produce high-quality patches. We study seven popular repair techniques applied to two large defect benchmarks of real-world defects.

We find that automated repair techniques are less likely to produce patches for defects that required developers to write a lot of code or edit many files, or that have many tests relevant to the defect, and that Java techniques are moderately more likely to produce patches for high-priority defects. The time it took developers to fix a defect does not correlate with automated repair techniques' ability to produce patches. A test suite's coverage also does not correlate with the ability to produce patches, but higher coverage correlated with higher-quality patches. Finally, automated repair techniques had a harder time fixing defects that required developers to add loops or new function calls, or change method signatures.

We produce a methodology and data that extend the ManyBugs and Defects4J benchmarks to enable evaluating new automated repair techniques, answering questions

such as "can automated repair techniques repair defects that are hard for developers to repair, or defects that developers consider important?"

Our findings both raise concerns about automated repair's applicability in practice, and also provide promise that, in some situations, automated repair can properly patch important and hard defects. Recent work on evaluating repair quality [29, 67, 184, 227, 252] has led to work to improve the quality of patches produced by automated repair [122, 175, 177]. Our position is that our work will similarly inspire new research into improving the applicability of automated repair to hard and important defects.

While it is less likely for repair techniques to fix hard or complex bugs as indicated in the findings, even for the relatively less complex bugs, the quality of patches produced by many APR techniques are often of low quality [252] and not semantically equivalent to developer-written patches [227]. This both raises an important concern about the practical usability of modern APR techniques, and drives research toward building techniques that produce higher-quality patches as described in the next chapter of this dissertation.

The work described in this chapter is joint with Sandhya Sankaranarayanan, René Just, and Yuriy Brun, and credit for this work is shared by all the researchers. Our extensions to the ManyBugs and Defects4J benchmarks and the scripts we have used to automate deriving the data for those benchmarks are available at `https://github.com/LASER-UMASS/AutomatedRepairApplicabilityData/`. The published version of this study [199] is available at `http://dx.doi.org/10.1007/s10664-017-9550-0`.

CHAPTER 3

QUALITY OF AUTOMATIC PROGRAM REPAIR ON REAL-WORLD DEFECTS

3.1 Introduction

APR techniques typically start with a defective program version and a set of passing and failing tests, and then modify the program version until finding a patch that makes all the tests pass. The underlying issue is that the set of tests provides a partial specification of the desired behavior, and thus the produced patches may overfit to those tests. This is known as the *patch overfitting* problem. For example, while, typically, many patches in a technique's search space pass the supplied tests, relatively few are equivalent to the developer-written patch [177, 227]; the APR technique has no way of knowing which is the better patch to return.

To address the patch overfitting problem, we first need a method to evaluate the quality of the produced patches. Prior studies of quality of APR have either used manual inspection [184, 227], or have used automatically generated, independent, evaluation test suites not used during the repair process [252, 307]. The issue with manual inspection is that it cannot scale to evaluate hundreds of automatically produced patches and can be subject to subconscious bias, especially if the inspectors are authors of the tools being evaluated [141]. Contrastingly, using evaluation tests is inherently partial, as the generated tests may undertest the patched program. Existing studies that use evaluation tests focus on small programs and relatively-easy-to-fix defects [252, 307].

This chapter describes our work that overcomes three considerable new engineering challenges. First, employing the objective, independent-test-suite-based evaluation of

patch quality, requires the creation of high-quality, automatically-generated test suites for real-world projects. We develop a methodology for using today's state-of-the-art test-suite generation techniques and overcoming their shortcomings to produce high-quality suites, and we release both the methodology and the generated test suites. Second, many APR techniques are designed and implemented for C (e.g., GenProg and TrpAutoRepair) and Par, designed and implemented for Java, was never released. We build JaRFly, the Java Repair Framework, which simplifies the implementation of Java techniques for genetic improvement (including but not limited to genetic improvement techniques for program repair), and release Java-based implementations of GenProg, Par, and TrpAutoRepair. Our implementations of GenProg and TrpAutoRepair are the first that faithfully follow the original techniques' designs, improving prior attempts at replicating these techniques for Java. Our release of the Par [123] implementation is the first ever public release of Par. JaRFly is the first framework of its kind that can handle the entire Defects4J dataset, including the Closure compiler subject program. Third, while semantic code search can produce high-quality patches [122], such an approach has never been demonstrated on real-world programs. We have designed SOSRepair, a novel approach to using semantic code search to repair programs, focusing on extending expressiveness to that of real-world C programs and improving the search mechanism's scalability.

This chapter is organized as follows. Section 3.2 describes our methodology to evaluate APR quality. Section 3.3 describes the methods to analyze the factors that could affect repair quality. Section 3.4 describes the repair techniques and the real-world defects we use to evaluate repair quality and perform factor analyses. Section 3.5 describes the key findings in terms of the research questions. Section 3.6 discusses the implications of our results, suggests future directions for research, and describes the limitations of our choices of subject repair tools and defects. Section 3.7 addresses the threats to validity of our study and Section 3.8 summarizes our contributions.

3.2 Methodology to Evaluate Repair Quality

There are two established methods for evaluating quality of APR, using an independent test suite not used during the construction of the patch [29, 252], and manual inspection [184, 227], typically for equivalence with a developer-written patch (though manual inspection has been used to measure how maintainable the patches are [83] and how likely developers are to accept them [123]). The two methodologies are complementary. Intuitively, the methodology that uses an independent test suite is more objective, whereas manual inspection is more subjective and can be subject to subconscious bias, especially if the inspectors are authors of one of the techniques being evaluated. A recent study found that manual-inspection-based quality evaluation can be imprecise [141], while independent-test-suite-based quality evaluation is inherently partial, as the independent test is a partial specification. As a result, manual evaluation of quality can imprecisely label patches as correct and incorrect. The test-suite-based evaluation cannot be imprecise, but may be incomplete, potentially mislabeling some patches as correct but never labeling a correct patch as incorrect.

In this study, we select to use the test-suite-based quality evaluation method because (1) it is objective and reproducible in a fully automated manner, (2) can scale to complex, real-world defects in real-world systems, which are the focus of our work (whereas manual inspection would require using the projects' developers with intricate project knowledge). Since this methodology necessarily underestimates overfitting (it never labels a correct patch as incorrect) [141], our findings of overfitting are, at worst, conservative. Two independent recent studies [141, 307] have empirically demonstrated that our independent-test-suite-based methodology is more reliable and more objective than manual inspection.

To objectively measure the quality of a generated patch, we need two independent test suites that specify the desired behavior of the program being repaired. One test suite can be used by the repair techniques to produce a patch for a defect. The

second, independent test suite is called the evaluation test suite; this test suite is used to measure the patch's quality. To create the second test suite, for each defect, we generated test inputs using an off-the-shelf automated test input generator, and using the developer-repaired code as an oracle of correct behavior.

This repair-quality methodology is only effective if the evaluation test suite is of high-quality. Coverage is widely-used in industry to estimate test-suite quality [105]. Using statement-level code coverage as a proxy for test suite quality, our goal was to generate, for each defect, a high-coverage test suite, thus implying that a big portion of the functionality of the inspected class is being evaluated. Specifically, we focused on the statement coverage of the methods and classes modified by the developer-written patch and designed a test generation methodology aimed to maximize that coverage. Ideally, we want the evaluation test suite to have perfect coverage, but modern automated test generation tools cannot achieve perfect coverage on all large real-world programs, in part because of limitations of such tools such as possible infinite recursion in the creation process or impreciseness of method signatures such as Java generics [81]. Thus, we set as our goal to generate, for each defect, a test suite that achieves 100% coverage on all developer-modified methods, and at least 80% coverage on all developer-modified classes. The choice of coverage criteria is a compromise between a reasonable measure of covering all the developer changes and the modern automated test generation tools' ability to generate high-coverage test suites.

We used the developer-patched version of the code to generate the evaluation test suite because it guarantees that this test suite covers at least one way of repairing the defect. An alternative to using the defective version of the code would not provide such a guarantee. Our choice might cause the evaluation test suites to more accurately measure the quality of patches that are structurally similar to the human-written patches, and would bias that measurement more favorably toward patches whose

behavior agrees with the human-written patches. Future work could attempt to mitigate these concerns by combining test suites generated using multiple versions of the code, and by using alternate information for oracles, such as natural language specifications [22, 90, 197, 265], other implementations of the same specification [188], or even the unpatched version [296, 311], though each of those approaches would introduce its own limitations.

To measure the quality of a produced patch, we start with the defective code version, apply the patch to that code, and execute the generated evaluation test suite. We call the total number of tests executed in the evaluation test suite T_{total} and the number of tests the patched version passes T_{pass}. The quality of a patch is $\frac{T_{pass}}{T_{total}}$, as defined by prior work [252]. A patch that passes all the tests in the evaluation test suite has 100% patch quality.

We also measure the quality of the defective code version by executing the evaluation test suite prior to applying the patch. This allows us to identify the quality improvement due to the patch.

3.3 Analyzing Factors That Could Affect Repair Quality

This section describes the four factors we identify that could potentially affect the quality of the patches produced by repair techniques and the methodologies to analyze the association between the identified factors and repair quality.

3.3.1 Test-Suite Coverage and Size

Intuition suggests that higher coverage test suites used to produce patches should lead to better-quality patches. Prior work empirically supports this intuition for heuristics-based program repair [252]; however, that work approximated the test suite coverage using test suite size and was not on real-world defects. In this study, we use real-world defects, measure the actual statement-level code coverage instead of

an estimate or proxy, and control for confounding factors, such as test suite size, defects' project, and the number of failing tests. In fact, prior studies of test suites have identified test suite size as often a confounding factor [120]. For the Defects4J dataset, we found statistically significant weak positive correlation ($r = 0.14$) between test suite size and statement-level coverage of the developer-written test suite on the defective code version. This is consistent with the prior studies [120].

Methodology: To measure the relationship between test suite coverage and repair quality, we attempted to create subsets of the developer-written test suite of varying coverage while controlling for test suite size, number of failing tests, and the defects themselves. However, we found that there is very low variability in the coverage of the individual tests and so we could not control for the test suite size while varying coverage. Hence, we generate the subsets while controlling for the number of failing tests and defects. Since test suite coverage and test suite size are positively correlated, analyzing their association with repair quality individually would not be appropriate. Thus, we use multiple linear regression to identify the relationship between two explanatory variables (test suite coverage and test suite size) and a response variable (repair quality).

For each of the defects patched by the repair techniques, we created subsets of the developer-written test suite of varying coverage. Each subset contains all the tests that evidence the defect, and randomly selected subsets of the rest of the tests. We then used the repair techniques to produce patches using these test suite subsets, and then computed the quality of the patches produced for each defect using the automatically-generated evaluation test suites.

To generate the test suite subsets for each defect, we first compute the minimum and the maximum code coverage ratio of the developer-written test suite of that defect. The *minimum code coverage ratio* ($cov_{\min}$) of a developer-written test suite is the statement coverage on the defective code version of just those tests that fail on the

defective code version and pass on the developer-repaired code version. We include all of these tests in every subset we generate, so their coverage is the minimum possible coverage. The *maximum code coverage ratio* ($cov_{\max}$) is the statement coverage on the defective code version of the entire developer-written test suite (the largest possible subset). We then compute the potential test suite coverage variability as the difference between the minimum and the maximum: $\Delta_{cov} = cov_{\max} - cov_{\min}$. Defects whose $\Delta_{cov} < 25\%$ lack sufficient variability in statement coverage to be used in this study, and we discard them. We chose five target coverage ratios evenly spaced between the minimum and the maximum: $cov_{\min} + \frac{1}{5}\Delta_{cov}$, $cov_{\min} + \frac{2}{5}\Delta_{cov}$, $cov_{\min} + \frac{3}{5}\Delta_{cov}$, $cov_{\min} + \frac{4}{5}\Delta_{cov}$, and $cov_{\min} + \Delta_{cov} = cov_{\max}$.

We used these target ratios to create 25 distinct test suites, 5 for each of the targets. For each target ratio c, we attempted to create five distinct test suite subsets within a 5% margin of c. (Note that there are typically multiple ways to achieve even $cov_{\max}$ coverage.) Each of the five test suite subsets started with all tests that fail on the defective code version and pass on the developer-repaired code version. We then iteratively attempted to add a uniformly randomly selected passing test case, without replacement, one at a time, as long as it did not make the subset's coverage exceed the target by more than 5%, stopping if the subset's coverage was within 5% of the target. If we attempted to add a test 500 times and failed to reach the target, we stopped.

Finally, for each technique, we computed a multiple linear regression considering patch quality as the dependent variable and test suite coverage and size as independent variables.

3.3.2 Defect Severity

The number of failing tests that trigger the defect are likely to be proportional to the number constraints that repair techniques need to satisfy to generate a repair.

The goal of this analysis is to measure the effect of the number of failing tests in the test suite used for producing the patches on the quality of repair techniques.

Methodology: To measure the effect of the number of failing tests in the test suite used to guide repair, we selected those defects that had at least 5 failing tests in the developer-written test suite and for which we are able to create high-quality evaluation test suite (recall Section 3.2). For each of the defects, we created 21 test suites subsets. We did this by first computing five evenly distributed target sizes s: $\frac{1}{5}f$, $\frac{2}{5}f$, $\frac{3}{5}f$, $\frac{4}{5}f$, and f, where f is the number of failing tests in the developer-written test suite (rounding to the nearest integer). Then, for each s (except $s = f$), we created 5 test suite subsets by including every passing test from the developer-written test suite, and uniformly randomly sampling, without replacement, s of the failing tests. This created 20 test suite subsets. We also included the entire developer test suite as a representative of the $s = f$ target, for a total of 21 test suite subsets. We then used the four automated repair techniques to attempt to patch the defects using each of the test suite subsets. Our methodology controls for the number of passing tests, unlike the prior study [252].

Both patch quality and the number of failing tests in the test suite used to guide repair are continuous variables, so we measure the association between these two variables using the Pearson correlation coefficient. This is typical for measuring the linear relationship between two continuous random variables.

3.3.3 Test-Suite Provenance

Prior work has suggested that using automatic test generation might improve program repair quality by increasing the coverage of the test suite used to produce the repair [252, 296, 311]. Augmenting a developer-written test suite with automatically-generated tests requires an oracle that specifies the expected test outputs. The unpatched program can be used as that oracle [296, 311], but that enforces the

assumption that the patch should avoid changing any behavior not explicitly exhibited by the failing tests. Other implementations of the same specification could similarly be used as an oracle [188], but this is only possible when multiple implementations exist (e.g., if repairing a browser and the expected behavior can be observed in an independent browser implementation) and requires defects in the implementations to be independent, which is often not the case in practice [128]. Finally, oracles can perhaps be extracted from comments or natural language specifications, for example with Swami [197], Toradacu [90], Jdoctor [22], or @tComment [265].

However, an earlier study found that even when a perfect oracle exists, using automatically-generated tests for program repair resulted in much lower quality patches than using developer-written tests (about 50% vs. about 80% quality) on small, student-written programs [252]. Thus, the goal of this analysis is to evaluate the effectiveness of using automatically generated tests to produce high-quality patches using program repair techniques.

Methodology: In this experiment, we compared the patches generated using developer-written test suites to patches generated using the automatically generated test suites. To control for the differences in the defects, properly measuring the association between test suite provenance and patch quality should be done using defects that can be patched using both kinds of test suites. If the set of defects patched using developer-written test suites differs from the set of defects patched using the automatically-generated test suites (as was the case in the earlier study [252]), then the defects can be a confounding factor in the experiment. For example, it is possible that more of the defects patched using one of the types of test suites are easier to produce high-quality patches for, unfairly biasing the results.

We thus started with the defects for which at least one of the repair techniques was able to produce a patch when using the developer-written test suites to guide repair, and first discarded those defects for which the automatically generated test suites did

not evidence the defect. To evidence the defect, at least one test in the test suite has to fail on the defective code version. (By definition, all automatically-generated tests pass on the developer-patched version, since that version is the oracle for those tests.) We next executed the repair techniques on the defects for which automatically generated tests evidenced the defect. For each repair technique, we identified the set of defects that were patched both using developer-written and using automatically-generated test suites. We call these the *in-common* populations. Note that these populations are, potentially, different for each repair technique.

To compare the quality of the patches on the in-common patch populations, we use the non-parametric Mann-Whitney U test. We choose this test because the two populations may not be from a normal distribution. This test measures the likelihood that the two populations came from the same underlying distribution. We compute Cliff's delta's δ estimate to capture the magnitude and direction of the estimated difference between the two populations. We also compute the 95% confidence interval (CI) of the δ estimate.

3.3.4 Fault Localization Accuracy

Most program repair techniques use off-the-shelf automated FL techniques to identify suspicious program elements such as methods or statements. We hypothesize that a potential reason for the repair techniques to produce low-quality patches is the imprecise fault locations the repair techniques use to modify and produce patches. The goal of this analysis is to test this hypothesis.

Methodology: Instead of using an off-the-shelf automated FL technique, we manually specify the candidate buggy code region that repair technique should modify. This is also known as the perfect FL [166]. We use the location of the code the developer modified to patch the defect as its candidate buggy code region, simulating the developer suggesting where the repair technique should try to repair a defect. We

then compare the number of defects correctly patched by the repair technique when using perfect FL than when using an automated FL technique.

3.4 Subjects of Investigation

This section details the subjects we investigated, in terms of the APR techniques and the real-world defects that these repair techniques could patch.

3.4.1 Automatic Program Repair Techniques

We conduct this study using two fundamentally different kinds of APR techniques: (1) Section 3.4.1.1 describes the four heuristics-based Java repair techniques and our JaRFly framework that implements three of these techniques. (2) Section 3.4.1.2 describes SOSRepair, a semantics-based repair technique that we develop to patch defects in large, real-world C programs. We analyze the effect of test-suite coverage and size, defect severity, and test-suite provenance on repair quality (Sections 3.3.1–3.3.3) using the heuristics-based repair techniques. We analyze the effect of fault localization accuracy on repair quality (Section 3.3.4) using SOSRepair.

3.4.1.1 Heuristics-Based Repair Techniques

We chose the following four representative heuristics-based repair techniques for our analysis. There are many existing heuristics-based techniques, often with similar performance. However, an underlying theory of heuristics-based repair suggests that analysis of a set of these techniques should generalize to others [282].

1. **GenProg [147, 284]** uses a genetic programming heuristic [133] to search the space of candidate repairs. Given a buggy program and a set of tests, GenProg generates a population of random patches by using statistical fault localization to identify which program elements to change (those that execute only on failing test cases or on both failing and passing text cases), and selecting elements from

elsewhere in the program to use as candidate patch code. The fitness of each patch is computed by applying it to the input program and running the result on the input test cases; a weighted sum of the count of passed tests informs a random selection of a subset of the population to propagate into the next iteration. These patch candidates are recombined and mutated to form new candidates until either a candidate causes the input program to pass all tests, or a preset time or resource limit is reached. Because genetic programming is a random search technique, GenProg is typically run multiple times on different random seeds to repair a bug.

2. **Par [123]** performs search by applying 12 fix templates — automatic program editing scripts created based on the fix patterns identified from developer fixes — in the locations they can be applied that are also identified as likely faulty by statistical fault localization.

3. **SimFix [111]**, mines code patterns (similar to Par templates) from frequently occurring code changes from developer-written patches. Then, in the project with the defect SimFix is attempting to repair, SimFix identifies code snippets that are similar to the code SimFix has localized the defect to. SimFix defines similarity using structural properties, variable names, and method names. SimFix ranks the code snippets by the number of times the mined patterns have to be applied to the snippet to replace the buggy code. SimFix then selects the snippets (one at a time) from the ranked list of top 100 snippets, applies the pattern-based modifications to produce a candidate patch, and validates the patch against the purified failing tests created using a test purification technique [302]. While the original paper describes SimFix stopping once a patch that passes the test suite is found [111], the implementation [108] generates multiple patches which

pass at least one of the purified failing tests. In this study, we use all the found patches for our analyses.

4. **TrpAutoRepair [224]** uses random search instead of genetic programming to traverse the search space of candidate solutions. Instead of running an entire test suite for every patch, TrpAutoRepair uses heuristics to select the most informative test cases first, and stops running the suite once a test fails. TrpAutoRepair limits its patches to a single edit. It is more efficient than GenProg in terms of time and test case evaluations [224]. The same approach is also called RSRepair [225], and we refer to the original algorithm name in this study.

GenProg and TrpAutoRepair were originally designed and implemented for C and Par was designed and implemented for Java, but was never released. We build JaRFly, the Java Repair Framework, which simplifies the implementation of Java techniques for genetic improvement (including but not limited to genetic improvement techniques for program repair), and release Java-based implementations of GenProg, Par, and TrpAutoRepair. We next describe the details of JaRFly framework.

JaRFly: The Java Repair Framework. JaRFly is our open-source framework

http://JaRFly.cs.umass.edu/

for implementing techniques for automatic search-based improvement (or *genetic improvement*) of Java programs. Genetic improvement approaches reuse existing software as input to metaheuristic search. The search goal is to identify variants of

that input software that improve on the software according to some criterion (e.g., functionality, performance) [219].

JaRFly is publicly available to facilitate researchers and practitioners building search-based improvement approaches for Java programs. The implementation includes reimplementations of GenProg [147] and TrpAutoRepair [224] for Java (original releases of these tools were for C programs), and releases the first public reimplementation of Par [123].

JaRFly's novelty and utility lie in the way it decouples the fundamental components of metaheuristic search and allows developers to specify just those fundamental components, taking care of the rest of the approach implementation. These components include problem representation, fitness function, mutation operators, and search strategy [98]. JaRFly provides high-level extension points for each of these components, which differentiates it from prior frameworks that support implementing Java-based repair techniques [185].

JaRFly simplifies the process of implementing genetic improvement approaches for Java programs. JaRFly handles parsing Java programs into a specified representation, and metaheuristic search over variants within that representation using specified mutation operators, search strategy, and fitness function. JaRFly allows the user to specify these representations, mutation operators, search strategies, and fitness functions by selecting from a set of already implemented options, or by extending with new versions via explicit extension points.

JaRFly improves on prior frameworks that support implementing Java-based repair techniques [185] by making these fundamental components explicit and supporting their extensions explicitly, while also handling a wider range of Java programs. For example, JaRFly can operate over the Closure compiler subject program from the Defects4J dataset, whereas prior frameworks cannot [185]. We next detail JaRFly's four fundamental components of metaheuristic search.

Problem Representation. The first and perhaps most fundamental design choice in applying metaheuristic search to a software engineering problem is deciding how to represent the problem such that it is amenable to symbolic manipulation. The most common representation choice in genetic improvement applications is the *patch representation*, in which an individual candidate solution is represented as a variable-length sequence of edits to the original program [147, 148]. In addition to Java, variations of and improvements on this representation choice can target Python [3] and C [209, 210] programs. Prior to the development of the patch representation, genetic-programming-based program repair operated over problems represented as a fixed-length weighted path through the program represented as an abstract syntax tree [80, 284]; as is typical in metaheuristic search, representation choice influences search success and efficiency [148]. By making this representation an explicit choice, and extension point, JaRFly enables developers to both pay proper attention to the choice of representation and to evaluate multiple representation choices.

JaRFly's `Representation` interface exports functionality for manipulating and evaluating a candidate solution in the context of a search-based program improvement approach. This includes support for

1. querying variant-specific localization information,

2. evaluating fitness, such as serializing a variant to disk and compiling it, or running one or more test cases against a given variant, tasks common to most genetic improvement approaches, depending on fitness function, and

3. assessing the validity of and applying mutation operators to the particular variant.

To that end, JaRFly's `Representation` is parameterized by a mutation interface that provides functionality for editing arbitrary Java programs.

JaRFly provides prebuilt implementations of (1) an abstract superclass that supports caching and serialization of common representation-independent intermediate data, such as a fitness cache, and (2) a classic patch representation for program repair problems in Java. The currently-implemented patch representation is a variable-length list of indivisible mutation operators, such as "Insert statement S at location L"; mutating this representation adds a new edit to the end of the current variant. It is straightforward to implement other choices without requiring major refactoring of the framework. For example, Oliveira et al. [209, 210] propose a novel patch-based representation that decouples the fault, operator, and fix spaces, with implications for crossover (but no other components of the search strategy); this could be achieved for Java in our framework by specializing the present patch-based representation (specifically the `getGenome` method) and implementing the new crossover operators in dedicated methods in the `Population` module.

Fitness Function. Applying metaheuristic search to a software engineering problem requires a *fitness function* to determine the fitness of a variant. Thus, this function must operate on the representation. JaRFly makes the choice of the fitness function explicit.

The most typical fitness function in modern repair approaches is a weighted sum of the number of test cases passed by a program variant. Sampling can reduce the computational cost of this fitness function [78]. Alternative fitness functions for program repair typically combine test cases with another objective, such as in a multi-objective search strategy. These alternative objectives can include a variant's similarity to patches in a dataset of previous developer-written patches [145], or its intermediate semantic distance according to a set of learned invariants over intermediate program state [64, 78] or according to memory values [56] from either the original program or the rest of the population.

JaRFly provides an extensible, representation-agnostic `Fitness` module that, by default, implements and provides configuration options for multiple common fitness strategies from the genetic improvement literature. These strategies include test execution at different levels of JUnit granularity (individual JUnit method, or entire JUnit class), and configuration options for test sampling (including generational versus individual sampling, and a configurable sample rate), and test selection (sampled, heuristically modeled [224, 282], or test to first failure). JaRFly's `Fitness` interface is agnostic to the underlying testing methodology, so it is not limited to using JUnit for fitness calculation. `Fitness` provides, by default, the idea of a (potentially dynamically-updated) *test model*, supporting experiments and extensions focused on more intelligent test selection and prioritization. JaRFly, moreover, extends (in a non-default branch) `Fitness` to evaluate and provide additional values, such as an experimental diversity-based metric [64], in the context of a multi-objective search strategy (NSGA-II [57]) extended from the `Search` module. Other measures of fitness, such as via comparison to a historical dataset of patches [145], can similarly extend `Fitness.testFitness` for more specialized, non-test-driven metrics.

Mutation Operators. Metaheuristic search requires a set of manipulation operators applicable to the selected representation. JaRFly provides the `EditOperation` abstraction, parameterized by a rewriter engine that can modify arbitrary Java programs. JaRFly's default implementation uses the Eclipse JDT API to perform rewriting. An `EditOperation` is instantiated at a particular (abstract) `Location`, and may contain one or more abstract `Hole`s that need to be filled in with suitable code. For example, an `Append` operation can be instantiated at any statement in a Java location; it has a single `Hole` that must be filled in by a piece of code that may be appended there.

JaRFly implements all of the statement-level edit operations used by GenProg and TrpAutoRepair and all Par fix templates, including the optional ones from `https:`

//sites.google.com/site/autofixhkust/home/, not included in the original Par paper [123]. Both GenProg and TrpAutoRepair construct modifications by reusing code from elsewhere in the program under repair. The `Representation` enforces this type of modification, providing information on legal `Location`s and code bank code that can be used to fill in `Hole`s for a particular variant. Meanwhile Par uses 12 fix templates — automatic program editing scripts created based on the fix patterns identified from developer-written patches. As with the coarser-grained operations used by GenProg and TrpAutoRepair, the `Representation` provides the possible values to fill in `Hole`s in Par's fix templates, such as which variable should be checked for `null` in the null-check-insertion template.

Some `EditOperation`s cannot be applied at all `Location`s. For example, an `Append` operation cannot insert code that references out-of-scope variables, or the result will not compile. JaRFly creates `EditOperation`s via a helper `JavaEditFactory`, which queries a variant via its `Representation` interface for information to determine the edit's legality. JaRFly implements a set of static semantic checks that can identify edits that will be rejected by the compiler. Previous work demonstrated that static semantic checks improve efficiency in genetic programming repair for C programs [148]. Java's compiler is substantially stricter than most C compilers, requiring commensurately more complex static checks to avoid invalid mutations.

Although we use the released SimFix implementation for our experiments, the mutation operators considered by SimFix could be implemented further as abstractions or extensions of this paradigm. Mutation operators are typically associated with weights that inform their selection and application. In the default implemented algorithms, these weights are fixed throughout the search strategy. However, they are customizable by design, such as via a machine-learned model of edit frequency drawn from historical, developer-written patches [177, 256].

Search Strategy. The choices of representation and mutation operators represent the space of possible variants metaheuristic search can explore, and the choice of fitness function represents the objective shape of that search space. The *search strategy* defines the path through the space the metaheuristic search uses to optimize the objective.

Common search strategies include local search, random search, and genetic programming. JaRFly's `Search` interface provides a representation-agnostic extension point for implementing search strategies, and implements five strategies, to facilitate comparison and customization. The implemented strategies are a random search, a weighted brute force single-edit search, an oracle search, a genetic programming heuristic, and NGSA-II [57], a multi-objective evolutionary search strategy.

In addition to these four fundamental components of the metaheuristic search, JaRFly includes implementation and support for other common and important interfaces and utilities for search-based program modification:

Population Manipulation. JaRFly implements crossover and selection strategies common in source-level evolutionary program manipulation. The implemented crossover strategies include one-point crossover, uniform crossover [284], and crossback crossover (crossover with the original unmodified representation) [284]. The one implemented selection strategy is tournament selection with configurable tournament sizes. JaRFly contains extension points to make adding new crossover and selection operators straightforward and independent of representation. Additionally, JaRFly allows setting the proportional mutation rate as a top-level configuration option.

Localization and Code Bank Management. Fault and fix localization are common concerns in search-based program repair or improvement. JaRFly implements common weighted path localization with configurable path weights, facilities for reading in arbitrary localization data from a file, and an abstract class for implementing

alternative localization strategies [226]. JaRFly uses the JaCoCo coverage library to compute coverage for the purposes of fault localization [72].

These facilities support significant (but straightforward) customization and investig--ation of all elements of a meta-heuristic search technique for program transformation. Implementing different metaheuristic search strategies (regardless of the search goal) requires specialization of a single `Search` class; investigating or isolating the effect of particular search features (such as selection, crossover or mutation rate, or the numerous other parameters influencing the traversal strategy in a genetic algorithm) requires the specialization of single methods, or the modification of existing top-level configuration options. These choices enable significant ongoing experimentation and specialization of the *search* component of a search-based or genetic improvement program modification strategy, without requiring reimplementation or modification of how programs under modification are represented, manipulated, or evaluated.

3.4.1.2 SOSRepair: Semantics-Based Repair Technique

We designed a novel repair technique called SOSRepair which builds on the underlying principles of SearchRepair [122], a semantics-based repair technique that is able to produce high-quality patches (patches passed 97.3% of independent, evaluation tests not used during patch construction) for bugs in small 24-line student-written C programs but could not be run on large programs. We fundamentally redesigned SearchRepair to create SOSRepair which is able to produce high-quality patches for large, real-world C programs. Appendix C describes the details of the SOSRepair approach and implementation.

3.4.2 Defect Benchmarks

To evaluate the quality of the four heuristics-based Java repair techniques we use Defects4J (v1.1.0) [119] benchmark, which consists of 357 defects taken from five real-world, open-source Java projects (recall Section 2.3.2).

To evaluate the quality of SOSRepair, we use the ManyBugs [149] benchmark, which consists of 185 defects taken from nine large, open-source C projects (recall Section 2.3.2). SOSRepair's is applicable to the 65 of the 185 ManyBugs defects (for details see Section C.4.1.2 in Appendix C).

3.5 Evaluating Repair Quality and Key Findings

In this section, we first summarize the key findings of this study in terms of the research questions we ask. Next, we describe the key findings of statistically analyzing factors that could affect repair quality. Finally, we describe the results that lead to the summarized findings.

We ask the following eight research questions.

RQ1 Do heuristics-based repair techniques produce patches for real-world Java defects?

Answer: Yes, although less often than for C defects. Overall, at least one of the four techniques produced at least one patch for 106 (29.7%) out of the 357 defects available in the Defects4J benchmark.

RQ2 Does SOSRepair produce patches for real-world C defects?

Answer: Yes, SOSRepair produces patches for 22 (34%) of the 65 defects available in the ManyBugs benchmark on which SOSRepair is applicable.

RQ3 How often and how much do the patches produced by heuristics-based techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification?

Answer: Often. For the four techniques we evaluated, only between 13.8% and 41.6% of the patches pass 100% of an independent test suite. Patches typically break more functionality than they repair.

RQ4 How often and how much do the patches produced by SOSRepair overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification? How does this compare with the state-of-the-art?

Answer: Often. While SOSRepair patches more defects than prior techniques, the repair quality of SOSRepair is comparable to the quality of other state-of-the-art repair techniques as SOSRepair patches produced for only 9 (41%) of the 22 defects pass all the tests in the evaluation test suite.

RQ5 How do the coverage and size of the test suite used to produce the patch affect patch quality?

Answer: Larger test suites produce slightly higher-quality patches, though, surprisingly, the effect is extremely small. Also surprisingly, higher-coverage test suites correlate with lower quality, but, again, the effect size is extremely small.

RQ6 How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit?

Answer: The number of failing tests correlates with slightly higher quality patches.

RQ7 How does the test suite provenance (whether it is written by developers or generated automatically) influence patch quality?

Answer: Test suite provenance has a significant effect on repair quality, although the effect may differ for different techniques. In most cases, human-written tests lead to higher-quality patches.

RQ8 How does FL accuracy affect the quality of patches produced by SOSRepair?

Answer: FL accuracy can significantly improve repair quality. Manually improving FL accuracy significantly improved (9 (41%) vs. 16 (70%)) the quality of the patches

produced by SOSRepair and also enabled it to patch one more defect (22 (34%) vs. 23 (35%)).

In the following sections, we describe the details of the results obtained for each of the above research questions.

3.5.1 Repairability: Ability to Produce a Patch

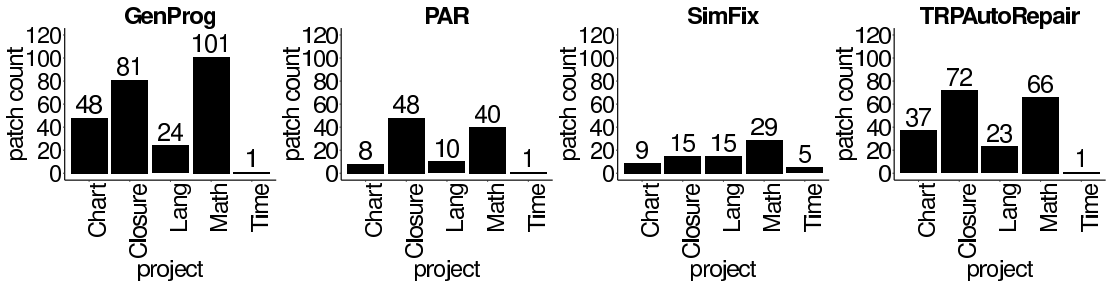
RQ1: Do heuristics-based repair techniques produce patches for real-world Java defects?

Figure 3.1(a) reports the results of the repair attempts made by the four heuristics-based techniques. GenProg patches 49 out of 357 defects (6 Chart, 15 Closure, 9 Lang, 18 Math, and 1 Time) and produces a total of 585 patches, out of which 255 are unique. Par patches 38 out of 357 defects (3 Chart, 12 Closure, 7 Lang, 15 Math, and 1 Time), and produces a total of 288 patches, out of which 107 are unique. SimFix patches 68 out of 357 defects (8 Chart, 15 Closure, 13 Lang, 27 Math, and 5 Time) and produces a total of 76 patches, out of which 73 are unique. TrpAutoRepair patches 44 out of 357 defects (7 Chart, 12 Closure, 8 Lang, 16 Math, and 1 Time) and produces a total of 513 patches, out of which 199 are unique. Overall, at least one technique produced at least one patch for 106 out of the 357 defects. All techniques produced at least one patch for 12 defects. SimFix most often produced patches (21.3% of the attempts) and produced patches for the most defects (19.0%). Figure 3.1(b) shows the distributions of unique patches, per project, generated by each of the four techniques.

Compared to prior studies on C defects [252], [149,224], the Java repair mechanisms produce patches on fewer repair attempts and for fewer defects. On C defects, GenProg produced patches for between 47% (ManyBugs defect dataset) and 60% (IntroClass defect dataset) and TrpAutoRepair produced patches for between 52% (ManyBugs) and 57% (IntroClass) defects. It is not surprising that on real-world defects, the rate is lower. Our findings are also consistent with prior work applying heuristics-based

technique	patches total		patches unique	defects patched
GenProg	585	(8.2%)	255	49 (13.7%)
Par	288	(4.0%)	107	38 (10.6%)
SimFix	76	(21.3%)	73	68 (19.0%)
TRPAutoRepair	513	(7.2%)	199	44 (12.3%)
total	1,462	(6.7%)	634	106 (29.7%)

(a) Produced patches



(b) Unique patch distributions, per technique

Figure 3.1: Repairability of heuristics-based techniques on real-world Java defects. (a) GenProg, Par, SimFix, and TrpAutoRepair produce patches 1,462 times (6.7%) out of the 21,777 attempts. At least one technique can produce a patch for 106 (29.7%) of the 357 real-world defects. (b) The distributions of unique patches produced by the four techniques are similarly shaped.

repair to Java defects, which found techniques to produce patches for 9.8%–15.6% of the defects [184]. In a prior study on Java defects, Par produced patches for 22.7% of the defects [123]. While that study's defects also came from real-world software projects, it is possible that the complexity of Defects4J defects results in the lower patch rates for Par. Some of the prior study's defects came from Lang and Math, projects that are also part of Defects4J (though a different set of defects), and our results on those projects are similar to those in the prior study [123]. Even though SimFix patches more defects (19.0%) than other techniques, the fraction of defects patched by SimFix is still much lower (19.0% vs. 47%) than that those obtained using repair techniques for C defects.

We conclude that heuristics-based techniques do produce patches on real-world Java defects, though the rate of patch production is lower than on C defects.

RQ2: Does SOSRepair produce patches for real-world C defects?

The first four columns of Figure 3.2 show the project, size of source code, number of developer-written tests, and the number of defective versions of the ManyBugs programs we use to evaluate SOSRepair. These defects require developers to edit one or more consecutive lines of code in a single location. As shown, SOSRepair patches 22 of the 65 defects.

program	kLOC	tests	defects	patched
`gmp`	145	146	2	0
`gzip`	491	12	4	0
`libtiff`	77	78	9	8
`lighttpd`	62	295	5	1
`php`	1,099	8,471	39	9
`python`	407	355	4	2
`wireshark`	2,814	63	2	2
total	5,095	9,420	65	22

Figure 3.2: Repairability of SOSRepair on real-world C defects. The table shows the programs in the ManyBugs benchmark that contain defects on which SOSRepair is applicable and the number of each for which SOSRepair generates a patch.

We conclude that SOSRepair produces patches for 22 (34%) of the 65 defects available in the ManyBugs benchmark on which SOSRepair is applicable..

3.5.2 Repair Quality

RQ3: How often and how much do the patches produced by heuristics-based techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification?

To analyze the quality of patches produced by heuristics-based techniques, we use EvoSuite [81] to generate high-quality, held-out evaluation test-suites for the 106 Defects4J defects patched by these techniques (recall Section 3.5.1) following the methodology described in Section 3.2. Figure 3.3 shows the distributions of the quality of the patches produced by each technique. Due to the nature of the space of possible patches, all four techniques produce the same patch for some defects, which, for example, caused the minimum exhibited quality patch to be identical for all four techniques. Overall, 74.1% of the patches (GenProg: 75.7%, Par: 86.2%, SimFix: 53.9%, and TrpAutoRepair: 80.5%), on average, failed at least one test, thus overfitting to the specification and failing to fully repair the defect. The mean quality of the patches varied from 95.7% to 96.4%. The relatively high fraction is not necessarily a proportional indication of the quality of repair: Defective code versions already pass 98.3% of the tests, on average, so a patch that passes 96.0% of the tests may not even be an improvement over the defective version.

Accordingly, next, we consider whether patches improve program quality. Figure 3.4 shows, for each of the patched defects, the change in the quality between the defective version and the patched version. A negative value implies that the patched version failed more evaluation tests than the defective version. When a technique produced multiple distinct patches for a defect, for this comparison, we used the highest-quality patch. For GenProg, 33.3% of the defects' patches improved the quality, 42.5% showed no improvement, and the remaining 24.2% decreased quality. For Par, 20.0% improved, 40.0% showed no improvement, and 40.0% decreased quality. For SimFix, 45.8% improved, 35.5% showed no improvement, and 16.7% decreased quality. For TrpAutoRepair, 32.3% improved, 25.8% showed no improvement, and 41.9% decreased quality. For Par and TrpAutoRepair, more patches broke behavior than repaired it, and the decrease in quality was, on average, larger than the improvement. For all the techniques, the majority (89 out of 137, 65.0%) of the patches decrease or fail to

technique	patch quality				100%-quality
	minimum	mean	median	maximum	patches
GenProg	64.8%	95.7%	98.4%	100.0%	24.3%
Par	64.8%	96.1%	98.5%	100.0%	13.8%
SimFix	65.0%	96.3%	99.9%	100.0%	46.1%
TrpAutoRepair	64.8%	96.4%	98.4%	100.0%	19.5%

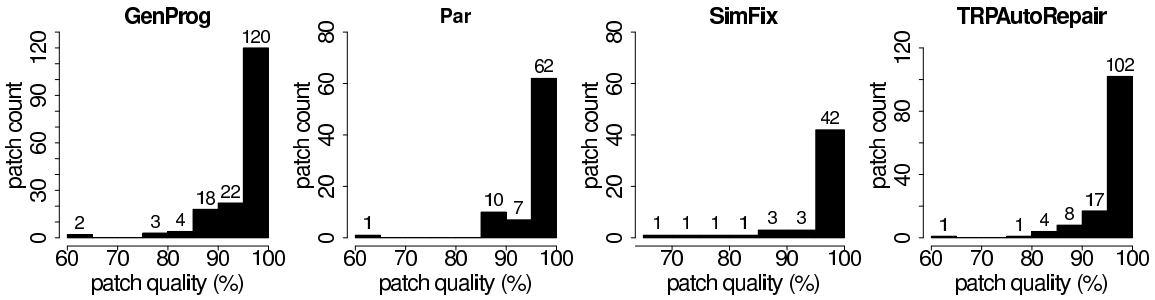
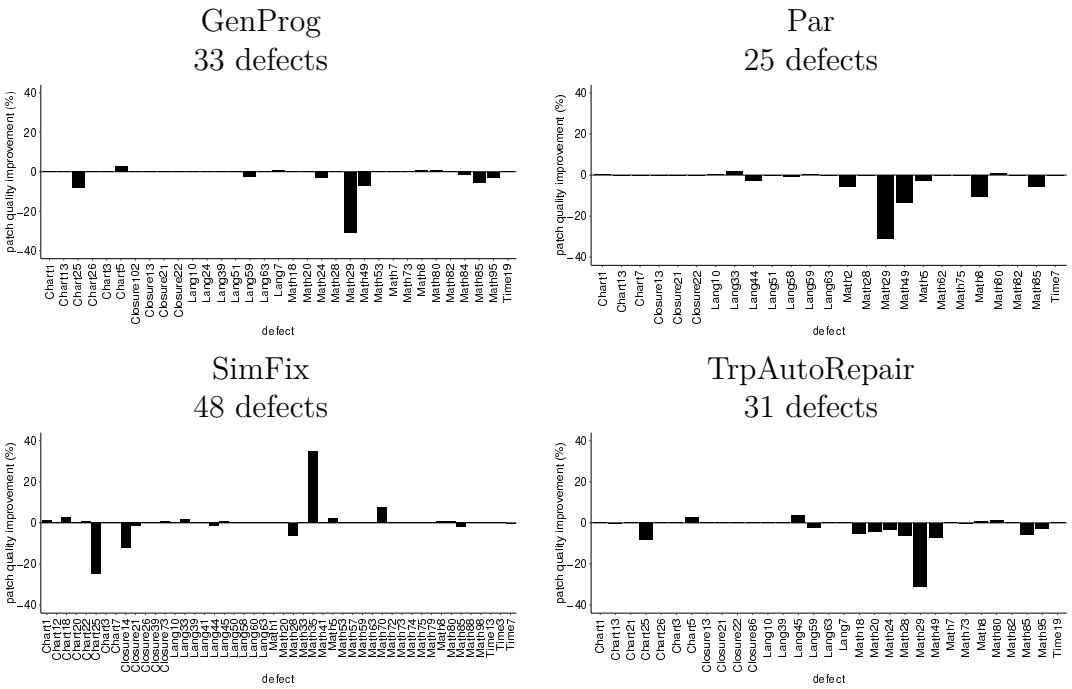


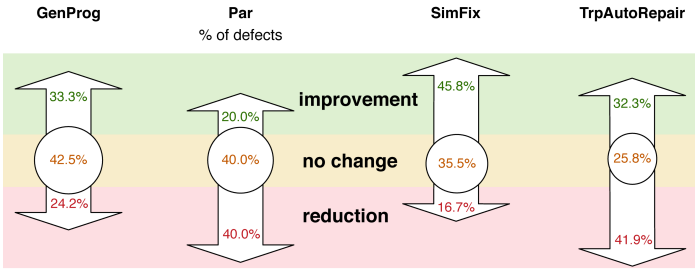
Figure 3.3: Repair quality of heuristics-based techniques on real-world Java defects. The quality of the patches the repair techniques generated when using the developer-written test suite varied from 64.8% to 100.0%. The distributions of patch quality is skewed toward the 100% end. On average, 74.1% (GenProg: 75.7%, Par: 86.2%, SimFix: 53.9% and Trp: 80.5%) of the patches failed at least one test.

improve quality, and more than a quarter (39 out of 137, 28.5%) of the patches break even more tests than they fix.

These results are consistent with the previous findings obtained using C repair techniques on small programs, where the median GenProg patch passed only 75% (mean 68.7%) of the evaluation test suite and the median TrpAutoRepair patch passed 75.0% of the evaluation test suite (mean 72.1%) [252].

We conclude that repair tool-generated patches on real-world Java defects often overfit to the test suite used in constructing the patch, often breaking more functionality than they repair.





technique	change in quality due to patch minimum	mean	median	maximum
GenProg	−30.9%	−1.7%	0.0%	2.6%
Par	−30.9%	−2.8%	0.0%	1.5%
SimFix	−24.9%	0.2%	0.0%	35.0%
TrpAutoRepair	−30.9%	−2.1%	0.0%	3.8%

Figure 3.4: Patch overfitting. Change in quality between the defective version and the patched version of the code. The median patch neither improves nor decreases quality. While more GenProg patches improve the quality than decrease it, the opposite is true for Par and TrpAutoRepair patches, and, on average, patches break more functionality than they repair. The data presented are for the 45 defects with high-quality evaluation test suites, of which GenProg produced patches for 33, Par for 25, and TrpAutoRepair for 31.

RQ4: How often and how much do the patches produced by SOSRepair overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification? How does this compare with the state-of-the-art?

To analyze the patch quality of SOSRepair and other APR techniques on ManyBugs defects, we create held-out evaluation test-suites using the following method. For a given defect, we automatically generate unit tests for all methods modified by either the project's developer or by at least one of the APR techniques in our evaluation. We do this by constructing small driver programs that invoke the modified methods:

- Methods implemented as part of an *extension* or *module* can be directly invoked from a driver's `main` function (e.g., the `substr_compare` method of `php string` module.)

- Methods implemented within *internal libraries* are invoked indirectly by using other functionality. For example, the method `do_inheritance_check_on_method` of `zend_compile` library in `php` is invoked by creating and executing `php` programs that implement inheritance. For such methods, the driver's `main` function sets the values of requisite global variables and then calls the functionality that invokes the desired method.

We automatically generate random test inputs for the driver programs that then invoke modified methods. We generate inputs until either the tests fully cover the target method or until adding new test inputs no longer significantly increases statement coverage. For four `php` and two `lighttpd` scenarios for which randomly generated test inputs were unable to achieve high coverage, we manually added new tests to that effect. For `libtiff` methods requiring tiff images as input, we use 7,214 tiff images randomly generated and released by the AFL fuzz tester [4]. We use the developer-patched behavior to construct test oracles, recording logged, printed, and

returned values and exit codes as ground truth behavior. If the developer-patched program crashes on an input, we treat the crash as the expected behavior.

Figure 3.5 shows the percent of evaluation tests passed by the SOSRepair. along with Angelix [190], Prophet [177], and GenProg [150], the state-of-the-art repair techniques we compare SOSRepair with. As shown, SOSRepair produces more patches (9, 41%) that pass all independent tests than Angelix (4), Prophet (5) and, GenProg (4). This suggests that semantic code search is a promising approach to generate high-quality repairs for real defects, and that it has potential to repair defects that are outside the scope of other, complementary repair techniques.

> We conclude that while SOSRepair patches more defects than prior techniques, the repair quality of SOSRepair is comparable to the quality of other state-of-the-art repair techniques as SOSRepair patches produced for only 9 (41%) of the 22 defects pass all the tests in the evaluation test suite.

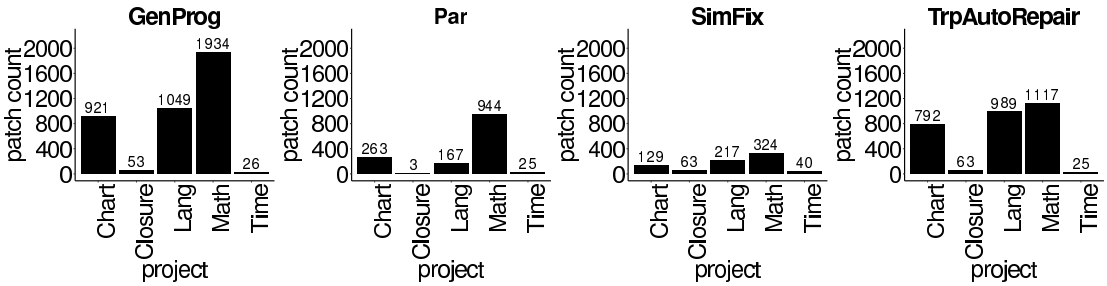
3.5.3 Test-Suite Coverage and Size

RQ5: How do the coverage and size of the test suite used to produce the patch affect patch quality?

We performed this analysis using the four heuristics-based repair techniques (Section 3.4.1.1) and the Defects4J benchmark (Section 3.4.2). Using the methodology described in Section 3.3.1, we were able to generate five distinct test suite subsets of varying coverage for 45 defects. For each of the 45 defects, we had 25 test suite subsets, and we attempted each repair 20 times using GenProg, Par, and TrpAutoRepair on different seeds, and one time using SimFix. In total, these 23,625 repair attempts produced 9,144 patches. Figure 3.6(a) shows the distribution of these patches. GenProg produced at least one patch for 29 out of the 45 defects, Par 25, SimFix 34, and TrpAutoRepair 29. (GenProg: 6 Chart, 2 Closure, 10 Lang, 10 Math, and, 1 Time; Par: 5 Chart, 1 Closure, 8 Lang, 10 Math, and, 1 Time; SimFix: 6 Chart, 3 Closure,

program ID:	revision pair	coverage	Angelix	Prophet	GenProg	SOSRepair	SOSRepair$^{\oplus}$
gmp-2:	14166-14167	—	✔	✘	✔	✘	✘
gzip-2:	3fe0caeada-39a362ae9d	—	✘	✔	✘	✘	✘
gzip-3:	1a085b1446-118a107f2d	—	✘	✘	✔	✘	✘
gzip-4:	3eb6091d69-884ef6d16c	—	✔	✘	✘	✘	✘
libtiff-1:	3b848a7-3edb9cd	90%	✔ 99%	✔ 99%	✔ 97%	✔ 97% ✂	✘
libtiff-2:	a72cf60-0a36d7f	76%	✔ 100%	✘	✔ 100%	✔ 100% ✂	✔ 100% ✂
libtiff-3:	eec4c06-ee65c74	73%	✔ 96%	✔ 96%	✔ 96%	✔ 96% ✂	✔ 96% ✂
libtiff-4:	09e8220-f2d989d	96%	✘	✘	✘	✔ 59% ⇜⇝	✔ 100% ✍
libtiff-5:	371336d-865f7b2	50%	✔ 100%	✘	✔ 98%	✔ 99% ⇜⇝	✘
libtiff-6:	764dbba-2e42d63	73%	✔ 92%	✔ 92%	✔ 28%	✔ 99% ✂	✔ 99% ⇜⇝
libtiff-7:	e8a47d4-023b6df	82%	✔ 100%	✘	✔ 0%	✔ 100% ✂	✔ 100% ⇜⇝
libtiff-8:	eb326f9-eec7ec0	90%	✘	✔ 100%	✔ 100%	✔ 60% ✍	✔ 100% ✍
libtiff-9:	b2ce5d8-207c78a	—	✘	✘	✘	✘	✘
lighttpd-1:	2661-2662	50%	NA	✔ 100%	✔ 100%	✔ 100% ✍	✔ 100% ✍
lighttpd-3:	2254-2259	—	NA	✘	✘	✘	✘
lighttpd-4:	2785-2786	—	NA	✘	✘	✘	✘
lighttpd-5:	1948-1949	—	NA	✔	✘	✘	✘
php-1:	74343ca506-52c36e60c4	89%	NA	NA	✔ 17%	✔ 100% ⇜⇝	✔ 100% ⇜⇝
php-2:	70075bc84c-5a8c917c37	86%	✔ 100%	✔ 100%	✔ 0%	✔ 100% ✂	✔ 100% ✂
php-3:	8138f7de40-3acdca4703	63%	‡	✔ 100%	✘	✔ 100% ✍	✘
php-4:	1e6a82a1cf-dfa08dc325	100%	NA	NA	✘	✔ 53% ✍	✔ 53% ✍
php-5:	ff63c09e6f-6672171672	79%	NA	NA	✔ 80%	✔ 90% ⇜⇝	✔ 50% ✍
php-6:	eeba0b5681-d3b20b4058	40%	NA	NA	✔ 0%	✔ 0% ✍	✔ 100% ⇜⇝
php-7:	77ed819430-efcb9a71cd	70%	✘	✔ 100%	✔ 50%	✔ 100% ✍	✘
php-8:	01745fa657-1f49902999	100%	✔ 67%	✘	✔ 100%	✔ 67% ⇜⇝	✘
php-9:	7aefbf70a8-efc94f3115	79%	NA	NA	✘	✔ 91% ⇜⇝	✘
php-14:	0a1cc5f01c-05c5c8958e	—	NA	NA	✔	✘	✘
php-15:	5bb0a44e06-1e91069eb4	—	✘	✔	✘	✘	✘
php-16:	fefe9fc5c7-0927309852	—	✘	✔	✔	✘	✘
php-17:	e65d361fde-1d984a7ffd	—	✔	✔	✘	✘	✘
php-18:	5d0c948296-8deb11c0c3	—	✘	✘	✘	✘	✘
php-19:	63673a533f-2adf58cfcf	—	✔	✔	✘	✘	✘
php-20:	187eb235fe-2e25ec9eb7	—	✔	✔	✔	✘	✘
php-21:	db01e840c2-09b990f499	—	✘	✘	✘	✘	✘
php-22:	453c954f8a-daecb2c0f4	—	✘	✘	✔	✘	✘
php-23:	b60f6774dc-1056c57fa9	—	✔	✔	✘	✘	✘
php-24:	1f78177e2b-d4ae4e79db	—	NA	NA	✘	✘	✘
php-25:	2e5d5e5ac6-b5f15ef561	—	NA	NA	✔	✘	✘
php-26:	c4eb5f2387-2e5d5e5ac6	—	NA	NA	✔	✘	✘
php-27:	ceac9dc490-9b0d73af1d	—	NA	NA	✔	✘	✘
php-28:	fcbfbea8d2-c1e510aea8	—	NA	NA	✘	✘	✘
php-29:	236120d80e-fb37f3b20d	—	NA	NA	✔	✘	✘
php-30:	55acfdf7bd-3c7a573a2c	—	NA	NA	✔	✘	✘
php-31:	ecc6c335c5-b548293b99	—	NA	NA	✔	✘	✘
php-32:	eca88d3064-db0888dfc1	—	NA	NA	✔	✘	✘
php-33:	544e36dfff-acaf9c5227	—	NA	NA	✘	✘	✘
php-34:	9de5b6dc7c-4dc8b1ad11	—	NA	NA	✔	✘	✘
php-35:	c1322d2505-cfa9c90b20	—	NA	NA	✔	✘	✘
php-36:	60dfd64bf2-34fe62619d	—	NA	NA	✔	✘	✘
php-37:	0169020e49-cdc512afb3	—	NA	NA	✘	✘	✘
php-38:	3954743813-d4f05fbffc	—	NA	NA	✘	✘	✘
php-39:	438a30f1e7-7337a901b7	—	NA	NA	✘	✘	✘
python-1:	69223-69224	100%	NA	✔ 33%	✘	✔ 76% ✍	✔ 76% ✍
python-2:	69368-69372	72%	NA	✔ 54%	✘	✔ 50% ⇜⇝	✔ 50% ✍
python-3:	70098-70101	—	NA	✔	✘	✘	✘
python-4:	70056-70059	—	NA	✘	✔	✘	✘
wireshark-1:	37112-37111	100%	✔ 87%	✔ 87%	✔ 87%	✔ 100% ✂	✔ 100% ✂
wireshark-2:	37122-37123	100%	NA	NA	✔ 87%	✔ 100% ✂	✔ 100% ✂
↓ additional defects patched by SOS$^{\oplus}$ ↓							
gmp-1:	13420-13421	97%	✔ 99%	✔ 99%	✘	✘	✔ 100% ✍
gzip-1:	a1d3d4019d-f17cbd13a1	79%	‡	✔ 100%	✘	✘	✔ 100% ✍
lighttpd-2:	1913-1914	56%	NA	✔ 100%	✘	✘	✔ 100% ⇜⇝
php-10:	51a4ae6576-bc810a443d	90%	NA	NA	✔ 92%	✘	✔ 92% ✍
php-11:	d890ece3fc-6e74d95f34	72%	‡	✔ 100%	✘	✘	✔ 100% ✍
php-12:	eeba0b5681-f330c8ab4e	42%	NA	NA	✔ 0%	✘	✔ 100% ✍
php-13:	80dd931d40-7c3177e5ab	71%	NA	NA	✘	✘	✔ 100% ✍

Figure 3.5: Repair quality of SOSRepair on real-world C defects. SOSRepair patches 22 of the 65 considered defects, 9 (41%) of which pass all of the independent tests. When manually provided a fault location (SOSRepair$^{\oplus}$), it patches 23 defects, 16 (70%) of which pass all of the independent tests. **Coverage** is the mean statement-level coverage of the independent tests on the patch-modified methods. ✔ means a patch was produced, ✘ means a produce could not be produced, and NA means the defect was not attempted by a technique. Three of the released Angelix patches [190] (denoted ‡) do not automatically apply to the buggy code.



(a) Distribution of patches generated using varying-coverage test suites.

technique	patch quality minimum	mean	median	maximum	100%-quality patches
GenProg	0.0%	94.8%	98.4%	100.0%	16.2%
Par	51.8%	91.2%	95.5%	100.0%	13.3%
SimFix	77.3%	98.4%	100.0%	100.0%	50.7%
TrpAutoRepair	62.9%	95.5%	99.0%	100.0%	19.0%

(b) Quality of patches generated using varying-coverage test suites.

technique	model quality p	R^2	test suite	p
GenProg	7.2×10^{-13}	0.013	size	6.7×10^{-13}
			coverage	8.5×10^{-4}
Par	5.2×10^{-12}	0.035	size	4.2×10^{-5}
			coverage	7.6×10^{-11}
SimFix	4.0×10^{-16}	0.086	size	2.7×10^{-7}
			coverage	1.3×10^{-15}
TrpAutoRepair	6.9×10^{-5}	0.0057	size	1.6×10^{-5}
			coverage	0.96

(c) Multiple linear regression relating coverage and size to patch quality.

Figure 3.6: The effect of test suite coverage and size on repair quality. (a) Distribution of the number of patches produced using developer-written test suite subsets of varying code coverage on the defective code version. (b) The quality of the patches generated using varying-coverage test suites varied from 0.0% to 100.0%. On average, 75.2% (GenProg: 83.8%, Par: 86.7%, SimFix: 49.3%, and TrpAutoRepair: 81.0%) of the patches failed at least one test. (c) A multiple linear regression reports that test suite size and test suite coverage are strongly significantly associated with patch quality ($p < 0.001$) except for coverage for TrpAutoRepair).

8 Lang, 13 Math, and 4 Time; and TrpAutoRepair 6 Chart, 2 Closure, 10 Lang, 10 Math, and, 1 Time.)

Figure 3.6(b) shows the statistics of the quality of the patches for those defects, created using the varying-coverage test suites. The quality varied, with GenProg even

producing some patches that failed *all* evaluation test cases. Overall, 75.2% of the patches, on average, failed at least one test in the evaluation test suite.

Next, for each technique, we created a multiple linear regression model to predict the quality of the patches based on the test suite coverage and size. Figure 3.6(c) shows, for each technique, the results of the regression model. All four fitted regression models are strongly statistically significant ($p < 0.001$) though with low R^2 values. Test suite size was a statistically significant predictor for patch quality for all four techniques, with larger test suites leading to higher-quality patches; however, with an extremely small effect size. Coverage was a less clear predictor: for TrpAutoRepair, the association was not statistically significant ($p > 0.1$), and was positive for GenProg and TrpAutoRepair, but negative for SimFix and Par. We further detail each technique's regression results next.

For GenProg, patch quality (on a 0–100 scale) is equal to $94.82 - 0.02(\text{coverage}) + 0.02(\text{size})$, where coverage is 100× the fraction of code in the defective code version covered by the test suite, and size is the normalized number of tests in the test suite used to generate the patch. Thus, the quality of the patch produced by GenProg decreases by 0.02% for each 1% increase in the test suite coverage and increases by 0.02% for each additional test in the test suite. While both associations of test suite coverage and size with the patch quality were statistically significant ($p < 0.001$), the magnitude is extremely small and the low R^2 value indicates little of the variability is explained. We conclude that test suite coverage and test suite size are significant predictors of patch quality, but the magnitude of the effect is extremely small, for GenProg.

For Par, the quality of the patch is equal to $91.18 - 0.10(\text{coverage}) + 0.03(\text{size})$. Thus, the quality of the patch produced by Par decreases by 0.10% for each 1% increase in the test suite coverage and increases by 0.03% for each additional test in the test suite. Again, while both associations of test suite coverage and test suite size

with patch quality are strongly statistically significant ($p < 0.001$), the magnitude is extremely small and the low R^2 value indicates little of the variability is explained. We conclude that both test suite coverage and test suite size are significant predictors of patch quality, but the magnitude of the effect is extremely small, for Par.

For SimFix, the quality of the patch is equal to $98.43 - 0.04(\text{coverage}) + 0.002(\text{size})$. Thus, the quality of the patch produced by SimFix decreases by 0.04% for each 1% increase in the test suite coverage and increases by 0.002% for each additional test in the test suite. We observe strongly statistically significant ($p < 0.001$) associations of test suite coverage and test suite size with patch quality however, the magnitude is extremely small and the low R^2 value indicates little of the variability is explained. We conclude that both test suite coverage and test suite size are significant predictors of patch quality, but the magnitude of the effect is extremely small, for SimFix.

For TrpAutoRepair, the quality of the patch is equal to $95.80 + 0.0003(\text{coverage}) + 0.006(\text{size})$. The equation implies that the patch quality of TrpAutoRepair increases by 0.0003% for 1% increase in the test suite coverage and increases by 0.006% for each additional test in test suite. The association of test suite size with patch quality is strongly statistically significant ($p < 0.001$), but that is not the case for test suite coverage. And, again, the magnitude of the association is extremely small and the low R^2 value indicates little of the variability is explained. We conclude that test suite size is a significant predictor of patch quality, but the magnitude of the effect is extremely small, for TrpAutoRepair.

We conclude that, surprisingly, both test suite size and test suite coverage have extremely small but statistically significant correlations with patch quality (positive for test suite size and negative for test suite coverage) produced using automatic program repair techniques.

3.5.4 Defect Severity

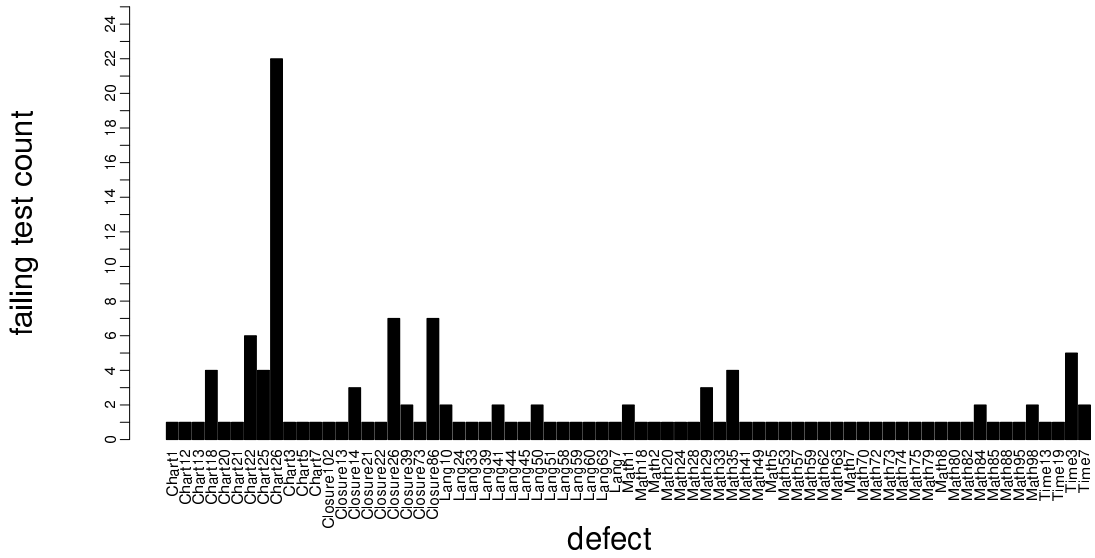
RQ6: How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit?

We performed this analysis using the four heuristics-based repair techniques (Section 3.4.1.1) and the Defects4J benchmark (Section 3.4.2). Figure 3.7(a) shows the frequency distribution of failing tests across the 71 defects for which at least one of the four techniques produced at least one patch, and for which we were able to create a high-quality evaluation test suite. Of these 71 defects, only 5 defects, Chart 22, Chart 26, Closure 26, Closure 86, and Time 3, have at least five failing tests.

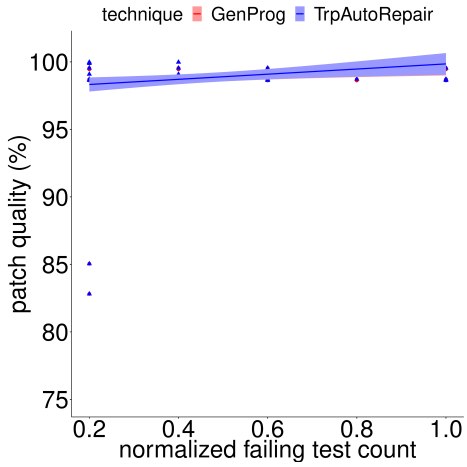
Figure 3.7(b) shows, for each technique, the quality of the patches produced, as a function of the fraction of the failing tests in the test suite used to guide repair. For GenProg and TrpAutoRepair, we observe statistically significant ($p < 0.05$) positive correlations (GenProg: $r = 0.18$, $p = 0.006$; TrpAutoRepair: $r = 0.19$ $p = 0.008$) between patch quality and the number of failing tests in the test suite. The 95% confidence interval for both techniques was $[0.05, 0.30]$.

Par did not produce any patches for any of the 5 defects considered for this analysis. Simfix only produced three patches and did not patch any of the 5 defects when using partial failing tests. Analyzing the execution logs of SimFix revealed that it was not able to localize the bug using partial failing tests. This suggests that fault localization strategy used by repair techniques could be a confounding factor when measuring the effect of the number of failing tests on patch quality. (Recall that SimFix and JaRFly use different fault localization techniques.)

> We conclude that the number of tests that a buggy program fails has a small but statistically significant positive effect on the quality of the patches produced using automatic program repair techniques and that this finding depends on the fault localization strategy used by the repair techniques.



(a)



(b)

Figure 3.7: The effect of the number of failing tests in test-suite on repair quality. (a) The distribution of the number of failing tests in the 71 defects for which at least one repair technique produces at least one patch and has a high-quality evaluation test suite. (b) Linear regression between patch quality and the number of failing tests and Pearson's correlation show statistically significant positive correlations for GenProg and TrpAutoRepair.

3.5.5 Test-Suite Provenance

RQ7: How does the test suite provenance (whether it is written by developers or generated automatically) influence patch quality?

We performed this analysis using the four heuristics-based repair techniques (Section 3.4.1.1) and the Defects4J benchmark (Section 3.4.2). In this experiment, we compared the patches generated using developer-written test suites to the patches generated using the EvoSuite-generated test suites. A technical challenge in executing repair techniques using EvoSuite-generated tests is a potential incompatibility between the bytecode instrumentation of EvoSuite tests with the bytecode instrumentation done by code-coverage-measuring tools employed by repair techniques for FL.

JaRFly uses JaCoCo [103] for fault localization and resolves instrumentation conflicts by updating the runtime settings of EvoSuite-generated tests (following official EvoSuite documentation[1]). The EvoSuite-generated tests are compatible with JaCoCo, Cobertura [43], Clover [13], and PIT [47] code coverage tools, but not with GZoltar [34]. Unfortunately, SimFix uses GZoltar, and so could not be included in this experiment. For GenProg, Par, and TrpAutoRepair, we used the developer-written patches as the oracle of expected behavior.

We thus started with the 68 defects for which at least one of the three repair techniques (GenProg, Par, and TrpAutoRepair) was able to produce a patch when using the developer-written test suites to guide repair. For 31 out of the 68 defects, automatically-generated test suites did not evidence the defect. This left 37 defects (5 Chart, 4 Closure, 11 Lang, 16 Math, and 1 Time). We next executed each of the three repair techniques on each of the 37 defects using the EvoSuite-generated test suites, using the methodology from Section 3.5.1, thus executing $37 \times 20 = 740$ repair attempts per technique. Note that comparing repair techniques' behavior with different test suites on these 37 defects is unfair because one of the criteria they satisfied to be selected is that at least one repair technique produced at least one patch for the defect using the developer-written test suite. Thus, for each technique,

[1] `http://www.evosuite.org/documentation/measuring-code-coverage/`

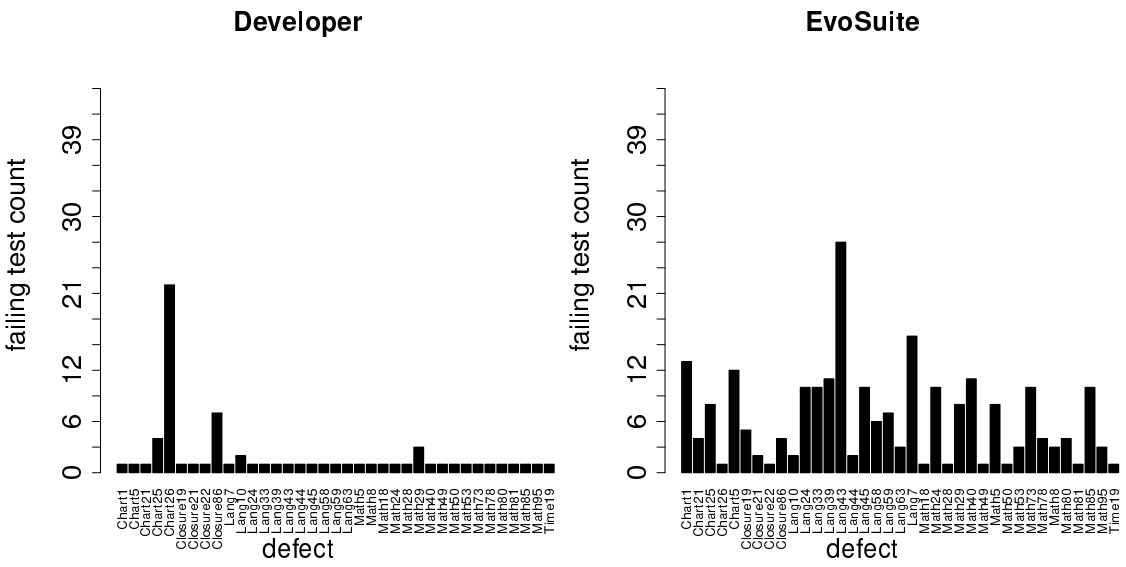
technique	test suite	generated patches	defects patched	patch quality min	mean	median	max	100%-quality patches
GenProg	developer	158 (21.4%)	29 (78.4%)	77.4%	94.9%	98.0%	100.0%	17.8%
	EvoSuite	98 (13.2%)	14 (37.8%)	6.3%	65.3%	54.3%	100.0%	8.2%
Par	developer	75 (10.1%)	20 (54.1%)	98.1%	98.4%	98.1%	99.7%	0.0%
	EvoSuite	17 (2.3%)	2 (5.4%)	97.2%	99.6%	99.9%	100.0%	41.2%
TrpAutoRepair	developer	128 (17.3%)	30 (81.1%)	77.4%	96.8%	98.1%	100.0%	24.6%
	EvoSuite	103 (13.9%)	17 (45.9%)	6.3%	65.2%	54.3%	100.0%	10.4%

Figure 3.8: The effect of test suite provenance on repairability. Using EvoSuite-generated tests, repair techniques were able to patch 37 of the the 68 defects.

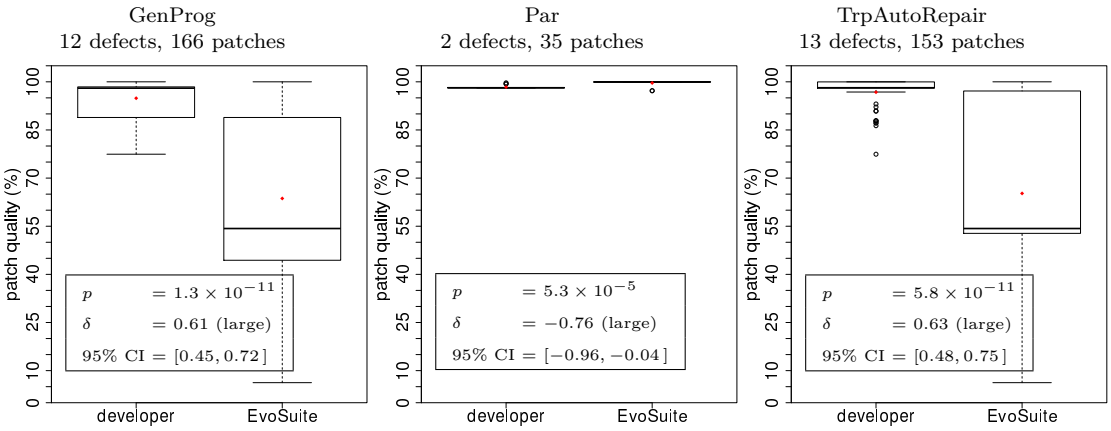
we identified the set of defects that were patched both using developer-written and using automatically-generated test suites. We call these the *in-common* populations. Note that these populations are, potentially, different for each technique.

Figure 3.8 and Figure 3.9 summarize our results. Figure 3.8 reports data for the 37 defects for which both test suites evidence the defect. As expected, because of the aforementioned bias in the selection of the 37 defects, using EvoSuite-generated test suites produced fewer patches and patches for fewer defects than using developer-written test suites. Using developer-written test suites produced a patch on between 10.1% and 21.4% executions, while using EvoSuite-generated test suites produced a patch on between 2.3% and 13.9% of the executions. Using developer-written test suites produced a patch for between 54.1% and 81.1% of the defects, while using EvoSuite-generated test suites produced a patch for between 5.4% and 45.9% of the defects.

In addition to the bias in defect selection, another possible reason that EvoSuite-generated test suites resulted in fewer patches could be differences in the test suites. Figure 3.9(a) shows the distributions of the number of failing (defect-evidencing) tests across the 37 defects for the two types of test suites. EvoSuite-generated test suites typically had more failing tests, perhaps contributing to it being more difficult to produce patches when using those test suites. Prior work has shown that having a larger number of failing tests correlated with lower patch production [199, 252].



(a) Distributions of failing tests in the 37 Defects4J defects' test suites.



(b) Patch quality comparison on the in-common (patched using both types of test suites) defect populations.

Figure 3.9: The effect of test suite provenance on repair quality. (a) The EvoSuite generated tests typically have more failing tests than the developer-written ones. (b) The box-and-whisker plots compare patch quality on the in-common defect populations, showing the maximum, top quartile, median, bottom quartile, and minimum values, with the mean as a red diamond. The patch quality of GenProg and TrpAutoRepair using the EvoSuite-generated tests is statistically significantly (Mann-Whitney U test) lower that those produced using developer-written tests. For Par, the effect is reversed. The δ estimate reports a large effect size and its 95% confidence intervals (CI) are entirely on one side of 0.

We compared the quality of the patches produced using the two types of test suites on the in-common populations. Figure 3.9(b) shows that for GenProg and TrpAutoRepair, the mean and median quality of the patches produced using the

developer-written test suites are higher than of those produced using EvoSuite-generated test suites. These differences are statistically significant (Mann-Whitney U test, $p = 1.3 \times 10^{-11}$ for GenProg, and $p = 5.8 \times 10^{-11}$ for TrpAutoRepair). The δ estimate computed using Cliff's delta shows a large effect size for the median patch quality of the patches produced using EvoSuite-generated test suites being lower for GenProg and TrpAutoRepair. The 95% CI does not spans 0 for both techniques, indicating that, with 95% probability, the two populations have different distributions.

For GenProg, this comparison is on the 12 in-common defects (Chart 5, Closure 22, Lang 43, Math 24, Math 40, Math 49, Math 50, Math 53, Math 73, Math 80, Math 81, and Time 19). On these defects, GenProg produced 73 patches using developer-written test suites and 93 patches using EvoSuite-generated test suites (166 patches total). For TrpAutoRepair, this comparison is on the 13 in-common defects (Chart 5, Closure 22, Closure 86, Lang 43, Lang 45, Math 24, Math 40, Math 49, Math 50, Math 73, Math 80, Math 81, and Time 19). On these defects, TrpAutoRepair produced 57 patches using developer-written test suites and 96 patches using EvoSuite-generated test suites (153 patches total).

Because the results for GenProg and TrpAutoRepair are derived from 12 and 13 defects, respectively, there is hope that these results will generalize to other defects. The same cannot be said for Par. Par produced patches using both types of test suites for only 2 out of the 37 defects (Closure 22 and Math 50). Figure 3.9(b) shows that the mean and median quality of the patches produced using the developer-written test suites are lower than those produced using EvoSuite-generated test suites. This result is statistically significant because Par produced 18 patches using developer-written test suites and 17 patches using EvoSuite-generated test suites, with $p = 5.3 \times 10^{-5}$ and the 95% CI interval does not span 0. However, while significant for these 2 defects, we cannot claim (nor do we believe that) this result generalizes to all defects from this 2-defect sample.

Our finding is consistent with the earlier finding [252] that provenance has a significant effect on repair quality, and that for GenProg and TrpAutoRepair, developer-written test suites lead to higher quality pathces. Surprisingly, the finding is opposite for Par (which was not part of the earlier study), with automatically-generated tests leading to higher-quality patches. Our study improves on the earlier work in many ways: We control for the defects in the two populations being compared, we use real-world defects, and we use a state-of-the-art test suite generator with a rigorous test suite generation methodology. The earlier study used a different generator (KLEE [33]) and aimed to achieve 100% code coverage on a reference implementation, but the generated test suites were small.

> We conclude that test suite provenance has a significant effect on repair quality, though the effect may differ for different techniques. For GenProg and TrpAutoRepair, patches created using automatically-generated tests had lower quality than those created using developer-written test suites. For a small, perhaps non-representative number of defects, Par-generated patches showed the opposite effect.

3.5.6 Fault Localization Accuracy

RQ8: How does FL accuracy affect the quality of patches produced by SOSRepair?

We performed this analysis using the SOSRepair (Section 3.4.1.2) and the 65 defects in ManyBugs benchmark on which SOSRepair is applicable (Section 3.4.2). We created SOSRepair$^{\oplus}$, a semi-automated version of SOSRepair that can take hints from the developer regarding fault location and variables of interest. SOSRepair$^{\oplus}$ differs from SOSRepair in the following two ways:

1. SOSRepair uses spectrum-based FL [117] to identify candidate buggy code regions. SOSRepair$^{\oplus}$ uses a manually-specified candidate buggy code region.

In our experiments, SOSRepair$^{\oplus}$ uses the location of the code the developer modified to patch the defect as its candidate buggy code region, simulating the developer suggesting where the repair technique should try to repair a defect.

2. SOSRepair considers all live variables after the insertion line in its query. While multiple mappings may exist that satisfy the constraints, not all such mappings may pass all the tests. SOSRepair uses the one mapping the SMT solver returns. SOSRepair$^{\oplus}$ can be told which variables not to consider, simulating the developer suggesting to the repair technique which variables likely matter for a particular defect. A smaller set of variables of interest increases the chance that the mapping the SMT solver returns and SOSRepair$^{\oplus}$ tries is a correct one. We found that for 6 defects (`gzip-1`, `libtiff-4`, `libtiff-8`, `php-10`, `php-12`, and `gmp-1`), SOSRepair failed to produce a patch because it attempted an incorrect mapping. For these 6 defects, we instructed SOSRepair$^{\oplus}$ to reduce the variables of interest to just those variables used in the developer's patch.

On our benchmark, SOSRepair$^{\oplus}$ patches 23 defects and 16 (70%) of them pass all independent tests. While it is unsound to compare SOSRepair$^{\oplus}$ to prior, fully-automated techniques, our conclusions are drawn only from the comparison to SOSRepair; the quality results for the SOSRepair$^{\oplus}$-patched defects for the prior tools in Figure 3.5 are only for reference.

Our experiments show that perfect FL allows SOSRepair$^{\oplus}$ to patch 7 additional defects SOSRepair could not (bottom of Figure 3.5), and to improve the quality of 3 of SOSRepair's patches. Overall, 9 new patches pass 100% of the independent tests.

We conclude that FL accuracy can significantly affect the quality of APR techniques. Manually improving FL significantly improved (9 (41%) vs. 16 (70%)) the quality of the patches produced by SOSRepair and enabled it to patch one more defect (22 (34%) vs. 23 (35%)).

3.6 Discussion

Our main finding is that patches produced by APR techniques often overfit to the tests used to produce those patches. The most important implication of our work is that research is needed into improving program repair techniques to produce higher-quality patches, or at least identifying and discarding lower-quality ones. Researchers can use the patch quality evaluation methodology and high-quality test suites we have developed to evaluate their techniques on real-world defects and demonstrate improvements over the state-of-the-art within this important dimension.

We observed that test-suite size correlates with higher-quality patches, and test-suite coverage correlates with lower-quality patches, though both effects are extremely small. Further, we found that human-written tests are, usually, better for APR than automatically-generated ones. These findings suggest that automatically generating tests to augment the developer-written tests may not help program repair. However, the method of generating the tests likely matters, and future research should study that relationship, in particular, exploring whether new approaches that generate tests from natural-language specifications [22, 197] are helpful.

Controlling for FL strategy, the number of tests a buggy program fails is positively correlated with higher-quality patches. On its face, this is surprising because fixing a larger number of failing tests usually requires fixing more behavior (although it is certainly possible for a small bug to cause many tests to fail, and for a large bug to cause only one test to fail). The key observation here is that FL can be a confounding factor. A larger number of failing tests can help FL identify the correct place to repair a defect, improving the chances the technique can produce a patch. In our study, we observe cases in which SimFix failed to localize a defect, and therefore failed to produce a patch when given fewer failing tests, but was able to do so with more failing tests (recall Section 3.5.4).

Finally, we observed that Java heuristics-based repair techniques produce patches for more defects than C heuristics-based repair techniques. Future research could target understanding the differences in the languages that cause this and improving the fix space and repair strategies used by the Java repair techniques.

Limitations. Research questions each impose specific requirements on the benchmark that can be used effectively to evaluate them. It is challenging for a single benchmark to satisfy these requirements for a diverse set of research questions, such as the ones we have explored in this study. For example, the majority of the Defects4J defects have a single failing test, which makes it hard to study the association between the number of failing tests and patch quality. Similarly, a lack of variability in the statement coverage of the developer-written tests makes it hard to study the relationships that involve that coverage. These shortcomings in the benchmark may reduce the strength of the results. Nevertheless, this study has developed a methodology that can be applied to other benchmarks to further study these questions.

JaRFly, our Java Repair framework, can help future researchers build new Java repair techniques. Our methodology for creating high-quality evaluation test-suites can be used to do so for new benchmarks, and the instances of evaluation test suites we have created for Defects4J can be used for future evaluations on that benchmark in a reproducible manner.

A recent study identified the evaluation-test-based approach to be reproducible, if conservative [141]: Evaluation test-suites may miss identifying some overfitting patches, but every patch they identify as overfitting, does so. This approach is complementary to manual inspection, which is less reliable but can identify some instances of overfitting that evaluation test suites miss [141]. Future research should pursue improving automated test generation with the goal of producing higher-quality evaluation test suites for program repair. Perhaps complementary to this challenge is

recent work on automatically generating test-suites from natural-language software artifacts (instead of human-patched version of code) [22, 197].

The generalizability of our results relies on the generalizability of the program repair techniques we use in our evaluation. While the classification of heuristics-based techniques [282] makes the argument that evaluations on representative techniques should generalize to other techniques in this class, evaluations on a larger, more diverse set of techniques provide stronger evidence. Applying our methodology to other techniques would constitute a valuable replication study. However, technological challenges prevented us from adding more techniques. Some projects do not release their tools' implementations, making reuse difficult. Some projects release only compiled binaries of their tools and do not make the source code public, which prevents minor modifications to those tool necessary for running experiments. For example, we were unable to use CapGen [287] in our evaluation because only its compiled binary is publicly available and we could not modify it to run using only a subset of the developer-written test-suites (as is required in Sections 3.3.1 and 3.3.2) and EvoSuite-generated test-suites (as is required in Section 3.3.3). Finally, some tools cannot be used as envisioned by the original project because of environmental changes. For example, we were unable to use ACS [298] in our evaluation because it was designed to work with a particular query style that directly interacts with GitHub, and GitHub has since disabled such queries. More generally, a recent empirical study on Java program repair techniques found that 13 out of the 24 (54%) techniques studied could not be used, including ACS and CapGen. The techniques could not be used because they were not publicly available, did not function as expected, required extraordinary manual effort to run (e.g., manual fault localization), or had hard-coded information to work on specific defect benchmarks and could not be modified with reasonable effort to work on others [66]. When possible, future research that produces

APR techniques should aim to make their tools public, releasing their source code, and avoid encoding specific benchmarks or experimental setups into the tools themselves.

3.7 Threats to Validity

Our study uses ManyBugs and Defects4J, well-established benchmark of defects in real-world, open source projects. The diversity, and real-world nature of these benchmarks mitigates the threat that our study will not generalize to other defects. Defects4J is evolving and growing with new projects, and our methodology can be applied to subsequently added projects, and to other benchmarks, to further demonstrate generalizability.

Our objective methodology for measuring patch quality requires independently generated test suites and the quality of those test suites affects our quality measurement. We use state-of-the-art automated test generation techniques, EvoSuite [81] and Randoop [212], but even state-of-the-art tools struggle to perform well on real-world programs. To mitigate this threat, we experimented with two test generation tools and their configuration parameters, developed a methodology for generating and merging multiple test suites, and only perform our study on the 71 out of 106 defects (67%) whose evaluation test suites met strict coverage criteria on the code affected by developer-written patches for the defects.

Our test-suite-based methodology to measure patch quality inherently overestimates the quality of patches because the evaluation test suites are necessarily partial specifications. If our methodology identifies a test that fails on a patch, the patch is necessarily incorrect; however, if our methodology deems a patch of 100% quality, there could still exist a hypothetical evaluation test the patch would fail. As a result, our conclusions are conservative. We find that APR often overfits on real-world defects, but the reality could be even more dire.

GenProg, Par, SimFix, and TrpAutoRepair are four representative heuristics-based automated program repair techniques. Prior work has explored similarity unifying heuristics-based repair and developed an underlying theory, suggesting that results from analysis of these four techniques should generalize to other heuristics-based techniques [282].

Our methodology follows the guidelines for evaluating randomized algorithms [10] and uses repair techniques' configuration parameters from prior evaluations that explored the effectiveness of those parameter settings [123, 147, 224]. We carefully control for a variety of potential confounding factors in our experiments, and use statistical tests that are appropriate for their context. We make all our code, test suites, and data public to increase researchers being able to replicate our results, explore variations of our experiments, and extend the work to other repair techniques, test suite generation tools, and defect datasets.

3.8 Contributions

We make the following contributions in this study:

- An empirical evaluation of quality of program repair on real-world Java defects, which outlines shortcomings and establishes a methodology and dataset for evaluating quality of new repair techniques' patches on real-world defects to promote research on high-quality repair. our generated test suites and experimental results are available from:

  `http://github.com/LASER-UMASS/JavaRepair-replication-package/`.

- A methodology for evaluating patch quality that fixes numerous shortcomings in prior work, properly controlling for potential confounding factors.

- A dataset of independent evaluation test suites for Defects4J defects, and a methodology for generating such test suites. Augmenting existing Defects4J

defects with two, independently created test suites can aid not only program repair, but other test-based technology.

- Java Repair Framework (`http://JaRFly.cs.umass.edu/`), a publicly released, open-source framework for building Java repair techniques, including our reimpl--ementations of GenProg [147], Par [123], and TrpAutoRepair [224]. JaRFly allows for easy combinations and modifications to existing techniques, and simplifies experimental design for automatic program repair on Java programs.

To make SOSRepair (Appendix C) possible, we make five major contributions to both semantic code search and program repair:

1. A more-scalable semantic search query encoding. We develop a novel, efficient, general mechanism for encoding semantic search queries for program repair, inspired by input-output component-based program synthesis [106]. This encoding efficiently maps the candidate fix code to the buggy context using a single query over an arbitrary number of tests. By contrast, SearchRepair [122] required multiple queries to cover all test profiles and failed to scale to large code databases or queries covering many possible permutations of variable mappings. Our new encoding approach provides a significant speedup over the prior approach, and we show that the speedup grows with query complexity.

2. Expressive encoding capturing real-world program behavior. To apply semantic search to real-world programs, we extend the state-of-the-art constraint encoding mechanism to handle real-world C language constructs and behavior, including structs, pointers, multiple output variable assignments, console output, loops, and library calls.

3. Search for patches that insert and delete code. Prior semantic-search-based repair could only *replace* buggy code with candidate fix code to affect repairs [122]. We extend the search technique to encode deletion and insertion.

4. Automated, iterative search query refinement encoding negative behavior. We extend the semantic search approach to include negative behavioral examples, making use of that additional information to refine queries. We also propose a novel, iterative, counter-example-guided search-query refinement approach to repair buggy regions that are not covered by the passing test cases. When our approach encounters candidate fix code that fails to repair the program, it generates new undesired behavior constraints from the new failing executions and refines the search query, reducing the search space. This improves on prior work, which could not repair buggy regions that no passing test cases execute [122].

5. Evaluation and open-source implementation. We release the implementation of SOSRepair (`https://github.com/squaresLab/SOSRepair`). We also release the independently-generated held-out evaluation test-suites for the ManyBugs defects used to evaluate SOSRepair and compare it with previous techniques (`https://github.com/squaresLab/SOSRepair-Replication-Package`).

The key findings presented in this chapter suggest that using better developer-written tests and accurate FL to guide the repair process in APR can significantly improve the repair quality. Based on these findings and the observation that most repair techniques only use test-suites for the repair process, I attempt to improve the repair quality by: (1) improving developer-written tests (Chapter 4) and (2) improving the accuracy of automated FL (Chapter 5) by deriving additional constraints from natural language software artifacts such as software specifications and bug reports.

The work described in this chapter is joint with Afsoon Afzal, Mauricio Soto, Yuriy Brun, Claire Le Goues, Kathryn T. Stolee, and René Just and credit for this work is shared by all the researchers. The published versions of the studies presented in this chapter [5, 200] can be found at `http://dx.doi.org/10.1109/TSE.2020.2998785` and `http://dx.doi.org/10.1109/TSE.2019.2944914`.

CHAPTER 4

IMPROVING DEVELOPER-WRITTEN TESTS USING NATURAL LANGUAGE SOFTWARE SPECIFICATIONS

4.1 Introduction

One of the key steps in the APR process is verifying that the patched program does what developer want it to do. Unfortunately, the most common way humans describe and specify software is natural language, which is difficult to formalize, and thus also difficult to use in an automated process as an oracle of what the software should do. Hence, repair techniques use developer-written test suites as a proxy for software specification. The developer-written tests are often inadequate [137] yet they are used by most APR tools because the tests are readily available and are machine-processable. Tests consist of two parts, an input to trigger a behavior and an oracle that indicates the expected behavior. While the state-of-the-art automated test generation techniques (e.g., Randoop [211], EvoSuite [81]) can effectively generate test inputs, they require a reference implementation to compute oracles for the generated inputs. In practice, a correct reference implementation may not be available, thus, limiting the use of such test generation techniques to improve the quality of APR tools. To address this, we analyzed other software artifacts from which we can derive the intended software behavior and improve the developer-written tests, which are used by APR tools. While formal, mathematical specifications that can be used automatically by computers are rare, developers do write natural language (NL) specifications, often structured (e.g., JavaDoc comments), as part of software requirements specification documents. Hence, in this chapter, we tackle the problem of automatically generating

tests from such structured NL specifications to verify that the software does what the specifications say it should.

We present an approach to automatically generate tests (inputs with oracles) from natural language specifications that can be used to verify that the software does what the specifications say it should. For example, Figure 4.1 shows a structured, natural language specification of a JavaScript `Array(len)` constructor (part of the official JavaScript specification ECMA-262 standard [290]) to be implemented in JavaScript implementations. Our approach focuses on generating oracles from such structured natural language specifications (test inputs can often be effectively generated randomly [81, 211], and together with the oracles, produce executable tests).

Of particular interest is generating tests for exceptional behavior and boundary conditions because, while developers spend significant time writing tests manually [8, 235], they often fail to write tests for such behavior. In a study of ten popular, well-tested, open-source projects, the coverage of exception handling statements lagged significantly behind overall statement coverage [90]. For example, Developers often focus on the common behavior when writing tests and forget to account for exceptional or boundary cases [8]. At the same time, exceptional behavior is an integral part of the software as important as the common behavior. An IBM study found that up to two thirds of production code may be devoted to exceptional behavior handling [49]. And exceptional behavior is often more complex (and thus more buggy) because anticipating all the ways things may go wrong, and recovering when things do go wrong, is inherently hard. Finally, exceptional behavior is often the cause of field failures [283], and thus warrants high-quality testing.

We present Swami, a technique for automatically generating executable tests from natural language specifications. We scope our work by focusing on exceptional and boundary behavior, precisely the important-in-the-field behavior developers often undertest [90, 283].

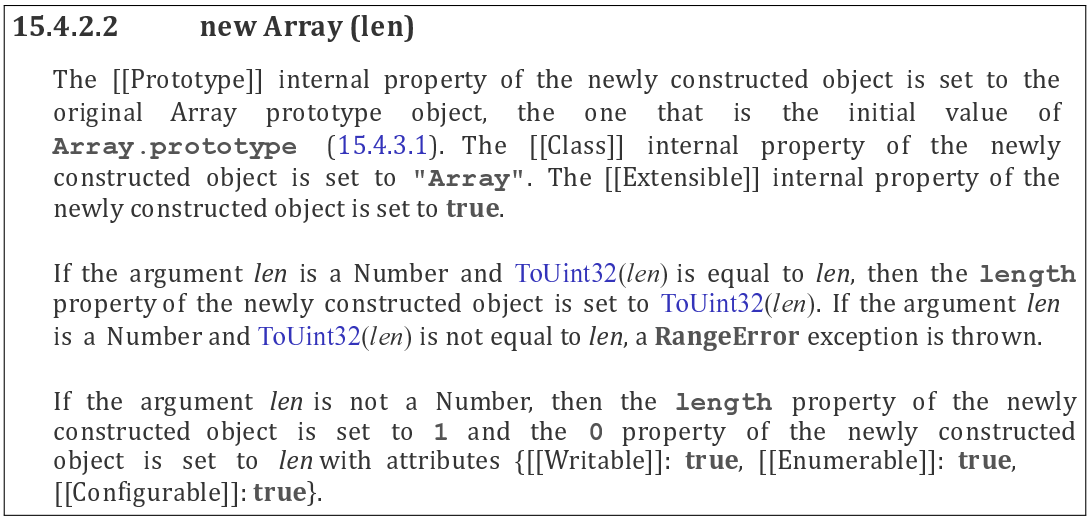
This chapter is organized as follows. Section 4.2 illustrates Swami on an example specification in Figure 4.1. Section 4.3 details the Swami approach. Section 4.4 describes the open-source specifications and software implementations we use to evaluate Swami and, Section 4.5 evaluates it. Section 4.6 compares Swami with the state-of-the-art. Section 4.7 describes future research directions. Section 4.8 discusses the threats to validity and Section 4.9 summarizes our contributions.

4.2 Intuition Behind Swami

To explain our approach for generating test oracles, we will use ECMA-262, the official specification of the JavaScript programming language [290]. The ECMA-262 documentation consists of hundreds of sections, including an index, a description of the scope, definitions, notational conventions, language semantics, abstract operations, and, finally, methods supported by the JavaScript language.

Our technique, Swami, consists of the following three steps: identifying parts of the documentation relevant to the implementation to be tested, extracting test templates from those parts of the documentation, and generating executable tests from those templates. We now illustrate each of these steps on the `Array(len)` constructor specification from ECMA-262 (Figure 4.1).

To use Swami to generate tests for a project, the developer needs to manually specify two things. First, a test constructor for instantiating test cases. For example, for Rhino, the test constructor is `new TestCase(test name, test description, expected output, actual output)`. This makes Swami project- and language-agnostic, for example, we were able to generate tests for Rhino and Node.js, two JavaScript implementations, one written in Java and one in C++, simply by specifying a different test constructor. Second, an implementation of abstract operations used in the specification. For example, the specification in Figure 4.1 uses the abstract operation `ToUnit32(len)`, specified in a different part of the specification document (Figure 4.5), and Swami



15.4.2.2 new Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of `Array.prototype` (15.4.3.1). The [[Class]] internal property of the newly constructed object is set to `"Array"`. The [[Extensible]] internal property of the newly constructed object is set to **true**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the `length` property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the `length` property of the newly constructed object is set to `1` and the `0` property of the newly constructed object is set to *len* with attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

`https://www.ecma-international.org/ecma-262/5.1/#sec-15.4.2.2`

Figure 4.1: Section 15.4.2.2 of ECMA-262 (v5.1), specifying the JavaScript `Array(len)` constructor.

needs an executable method that encodes that operation. For JavaScript, we found that implementing 10 abstract operations, totaling 82 lines of code, was sufficient for our purposes. Most of these abstract operation implementations can be reused for other specifications, so the library of abstract operation implementations we have built can be reused, reducing future workload.

4.2.1 Identifying Relevant Specifications

The first step of generating test oracles is to identify which sections of the ECMA-262 documentation encode testable behavior. There are two ways Swami can do this. For documentation with clearly delineated specifications of methods that include clear labels of names of those methods, Swami uses a regular expression to match the relevant specifications and discards all documentation sections that do not match the regular expression. For example, Figure 4.1 shows the specification of the `Array(len)` constructor, clearly labeling the name of that method at the top. This is the case for all of ECMA-262, with all specifications of methods clearly identifiable and labeled. For specifications like this one, Swami uses a regular expression that matches the

section number and the method name (between the section number and the open parenthesis).

When the documentation is not as clearly delineated, Swami can still identify which sections are relevant, but it requires access to the source code. However, to reiterate, this step, and the source code, are not necessary for ECMA-262, hundreds of other ECMA standards, and many other structured specification documents. Swami uses the Okapi information retrieval model [237] to map documentation sections to code elements. The Okapi model uses term frequencies (the number of times a term occurs in a document) and document frequency (the number of documents in which a term appears) to map queries to document collections. Swami uses the specification as the query and the source code as the document collection. Swami normalizes the documentation, removes stopwords, stems the text, and indexes the text by its term frequency and document frequency. This allows determining which specifications are relevant to which code elements based on terms that appear in both, weighing more unique terms more strongly than more common terms. For example, when using the Rhino implementation, this approach maps the specification from Figure 4.1 as relevant to the five classes listed in Figure 4.2. The 5th-ranked class, `NativeArray.java`, is the class that implements the `Array(len)` constructor, and the other four classes all depend on this implementation. Swami uses a threshold similarity score to determine if a specification is relevant. We empirically found that a similarity score threshold of 0.07 works well in practice: If at least one class is relevant to a specification document above this threshold, then the specification document should be used to generate tests. With this threshold, Swami's precision to identify relevant specification sections is 79.0% and recall is 98.9%, but that Swami can use regular expressions to remove improperly identified specifications, boosting precision to 93.1%, and all tests generated from the remaining improperly identified specifications will fail to compile and can be discarded automatically.

section ID	matched Java class	similarity score
15.4.2.2	`ScriptRuntime.java`	0.37
15.4.2.2	`Interpreter.java`	0.31
15.4.2.2	`BaseFunction.java`	0.25
15.4.2.2	`ScriptableObject.java`	0.24
15.4.2.2	`NativeArray.java`	0.21

Figure 4.2: Source files ranked based on their similarity with the specification in Figure 4.1 using Swami's information-retrieval-based approach. Using the information-retrieval-based approach to identify the documentation sections relevant to the implementation, Swami finds these Rhino classes as most relevant to section 15.4.2.2 of the ECMA-262 (v5.1) documentation (the specification of the `Array(len)` constructor from Figure 4.1). `NativeArray.java` is the class that implements the `Array(len)` constructor, and the other four classes all depend on this implementation. Each of the classes listed appears in the `org.mozilla.javascript` package, e.g., `ScriptRuntime.java` appears in `org.mozilla.javascript.ScriptRuntime.java`.

4.2.2 Extracting Test Templates

For the specification sections found relevant, Swami next uses rule-based natural language processing to create test templates: source code parameterized by (not yet generated) test inputs that encodes the test oracle. Swami generates two types of tests: boundary condition and exceptional condition tests. For both kinds, Swami identifies in the specification (1) the signature (the syntax and its arguments) of the method to be tested, the (2) expected output for boundary condition tests, and (3) the expected error for exceptional condition tests.

The `Array(len)` specification shows that the constructor has one argument: `len` (first line of Figure 4.1). The specification dictates that `Array(len)` should throw a `RangeError` exception if `len` is not equal to `ToUnit32(len)` (second paragraph of Figure 4.1), where `ToUnit32(len)` is an abstract operation defined elsewhere in the specification (Figure 4.5). We now demonstrate how Swami generates tests for this exceptional condition.

We have written four rules that Swami uses to extract this information. Each rule consists of a set of regular expressions that determine if a natural language sentence

satisfies the conditions necessary for the rule to apply, and another set of regular expressions that parse relevant information from the sentence. The rule also specifies how the parsed information is combined into a test template. Swami applies the four rules in series.

The first rule, *Template Initialization*, parses the name of the method being tested and its arguments. The rule matches the heading of the specification with a regular expression to identify if the heading contains a valid function signature i.e., if there exists a function name and function arguments. In the example specification shown in Figure 4.1, the top line (heading) "15.4.2.2 new Array(len)" matches this regular expression. Swami then creates an empty *test template*:

```
1 function test_new_array(len){}
```

and stores the syntax for invoking the function (`var output = new Array(len)`) in its database for later use.

The second rule, *Assignment Identification*, identifies assignment descriptions in the specification that dictate how variables' values change, e.g., "Let posInt be sign(number) × floor(abs(number))" in Figure 4.5. For assignment sentences, Swami's regular expression matches the variable and the value assigned to the variable and stores this association in its database. Here, Swami would match the variable `posInt` and value `sign(number)` × `floor(abs(number)`. This populates the database with variable-value pairs as described by the specification. These pairs will be used by the third and fourth rules.

The third rule, *Conditional Identification*, uses a regular expression to extract the condition that leads to returning a value or throwing an error. The rule fills in the following template:

```
if (<condition>){
  try{
     <function call>
  }
  catch(e){
     <test constructor>(true, eval(e instanceof <expected error>))
  }
}
```

replacing "`<function call>`" with "`var output = new Array(len)`", the method invoking syntax from the database (recall it being stored there as a result of the first rule). The `<test constructor>` is a project-specific description of how test cases are written. This is written once per project, by a developer.

The sentence "If the argument len is a Number and ToUnit32(len) is not equal to len, a RangeError exception is thrown" (Figure 4.1) matches this expression and Swami extracts the condition "argument len is a Number and ToUnit32(len) is not equal to len" and the expected error "RangeError exception is thrown", populating the following template, resulting in:

```
if (argument len is a Number and ToUnit32(len) is not equal to len
    ){
  try{
    var output = new Array(len)
  }
  catch(e){
    new TestCase("test_new_array_len", "test_new_array_len", true,
        eval(e instanceof a RangeError exception is thrown))
    test()
  }
}
```

Finally, the forth rule, *Conditional Translation*, combines the results of the previous steps to recursively substitute the variables and implicit operations used in the conditionals from the previous rule with their assigned values, until all the variables left are either expressed in terms of the method's input arguments, or they are function calls to abstract operations. Then, Swami embeds the translated conditional into the initialized test template, creating a test template that encodes the oracle from the specification. In our example, this step translates the above code into:

```
function test_new_array(len){
    if (typeof(len)=="number" && (ToUint32(len)!=len)){
        try{
          var output = new Array(len)
        }catch(e){
          new TestCase("test_new_array_len", "test_new_array_len",
              true, eval(e instanceof RangeError))
          test()
        }
    }
}
```

4.2.3 Generating Executable Tests

Swami uses the above test template with the oracle to instantiate executable tests. To do this, Swami generates random inputs for each argument in the template (`len`, in our example). ECMA-262 describes five primitive data types: Boolean, Null, Undefined, Number, and String (and two non-primitive data types, Symbol and Object). Swami generates random instances of the five primitive data types and several subtypes of Object (Array, Map, Math, DateTime, RegExp, etc.) using certain heuristics that help reduce fall alarms. We have found empirically that generating such inputs for all arguments tests parts of the code uncovered by developer written tests, which is consistent with prior studies [90]. Combined with the generated oracle encoded in the template, each of these test inputs forms a complete test.

Swami generates 1,000 random test inputs for every test template. Many of these generated tests will not trigger the generated conditional, but enough do. The bottom part of Figure 4.3 shows representative tests automatically generated for the `test_new_array` template.

Finally, Swami augments the generated test file with the manually implemented abstract operations. Figure 4.3 shows the final test file generated to test the `Array(len)` constructor.

```
/*ABSTRACT FUNCTIONS*/
function ToUint32(argument){
    var number = Number(argument)
    if (Object.is(number, NaN) || number == 0 || number == +0 ||
       number== -0 || number == Infinity || number == -Infinity){
        return 0
    }
    var i = Math.floor(Math.abs(number))
    var int32bit = i%(Math.pow(2,32))
    return int32bit
}
...
/*TEST TEMPLATE GENERATED AUTOMATICALLY*/
function test_new_array(len){
    if (typeof(len)=="number" && (ToUint32(len)!=len)){
        try{
          var output = new Array(len)
        }catch(e){
          new TestCase("test_new_array_len", "test_new_array_len",
              true, eval(e instanceof RangeError))
          test()
        }
    }
}
/*TESTS GENERATED AUTOMATICALLY*/
test_new_array(1.1825863363010669e+308)
test_new_array(null)
test_new_array(-747)
test_new_array(368)
test_new_array(false)
test_new_array(true)
test_new_array("V7KO08H")
test_new_array(Infinity)
test_new_array(undefined)
test_new_array(/[^.]+/)
test_new_array(+0)
test_new_array(NaN)
test_new_array(-0)
...
```

Figure 4.3: The executable tests automatically generated for the `Array(len)` constructor from the specification in Figure 4.1.

4.3 Swami Approach

Using natural language specifications pose numerous challenges. Consider the ECMA-262 specification of a JavaScript `Array(len)` constructor in Figure 4.1. The specification:

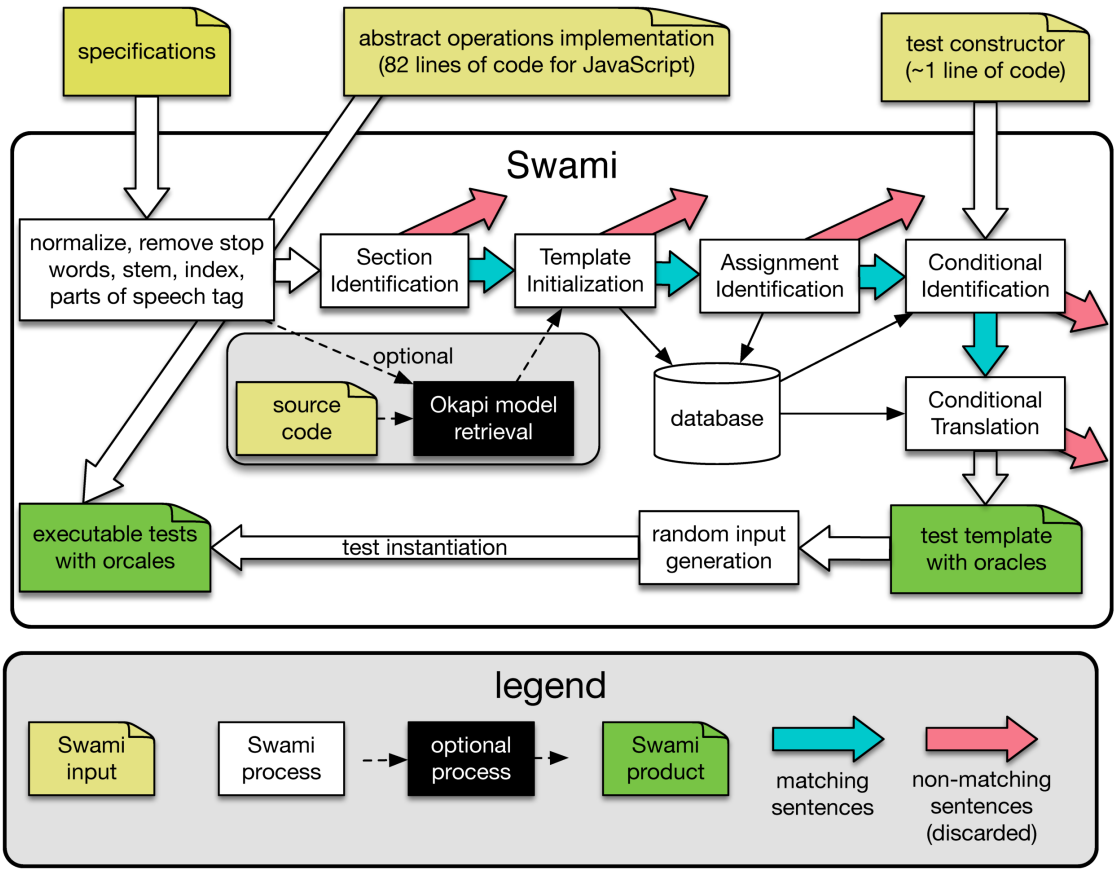
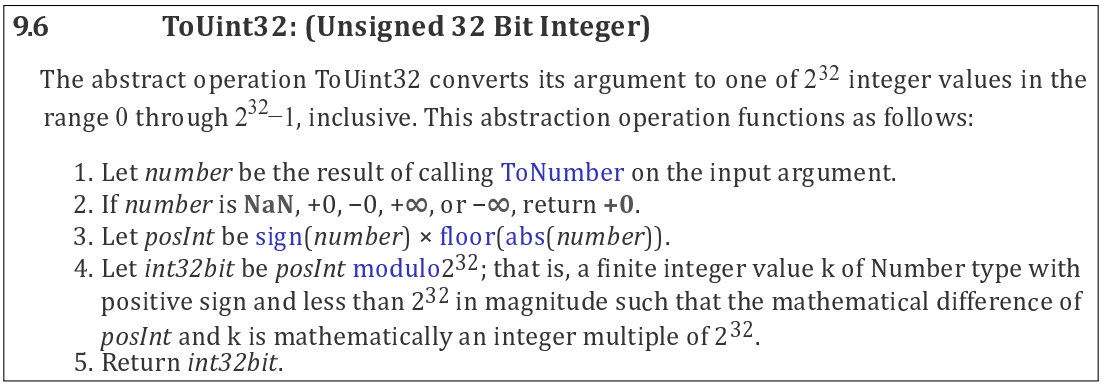


Figure 4.4: The Swami approach. Swami generates tests by applying a series of regular expressions to processed (e.g., tagged with parts of speech) natural language specifications, discarding non-matching sentences. Swami does not require access to the source code; however, Swami can optionally use the code to identify relevant specifications. Swami's output is executable tests with oracles and test templates that can be instantiated to generate more tests.

- Uses natural language sentences, such as "If the argument len is a Number and ToUint32(len) is equal to len, then the length property of the newly constructed object is set to ToUint32(len)."

- Refers to abstract operations defined elsewhere in the specification, such as `ToUint32`, which is defined in section 9.6 of the ECMA-262 specification document (Figure 4.5).



9.6 ToUint32: (Unsigned 32 Bit Integer)

The abstract operation ToUint32 converts its argument to one of 2^{32} integer values in the range 0 through $2^{32}-1$, inclusive. This abstraction operation functions as follows:

1. Let *number* be the result of calling ToNumber on the input argument.
2. If *number* is **NaN**, +0, −0, +∞, or −∞, return **+0**.
3. Let *posInt* be sign(*number*) × floor(abs(*number*)).
4. Let *int32bit* be *posInt* modulo 2^{32}; that is, a finite integer value k of Number type with positive sign and less than 2^{32} in magnitude such that the mathematical difference of *posInt* and k is mathematically an integer multiple of 2^{32}.
5. Return *int32bit*.

`https://www.ecma-international.org/ecma-262/5.1/index.html#sec-9.6`

Figure 4.5: An example abstract operation in the ECMA specification. ECMA specifications include references to abstract operations, which are formally defined elsewhere in the specification document, but have no public interface. Section 9.6 of ECMA-262 (v5.1) specifies the abstract operation `ToUnit32`, referenced in the specification in Figure 4.1.

- Refers to implicit operations not formally defined by the specification, such as `min`, `max`, `is not equal to`, `is set to`, `is an element of`, and `is greater than`.
- Implicitly uses local variables specified in assignment statements (e.g., *number*, *posInt*, and *int32bit* used in the specification shown in Figure 4.5) to describe oracles.
- Describes complex control flow, such as conditionals, using the outputs of abstract and implicit operations, and local variables in other downstream operations and conditionals.

Our technique, Swami, addresses these challenges and generates executable tests using the approach (Figure 4.4) described next. Swami normalizes and parts-of-speech tags the specifications (Section 4.3.1), identifies the relevant sections (Section 4.3.2), uses regular-expression-based rules to create a test template encoding an oracle (Section 4.3.3), and instantiates the tests via heuristic-based, random input generation (Section 4.3.4).

4.3.1 Specification Preprossessing

Swami uses standard natural language processing [118] to convert the specification into more parsable format. Swami normalizes the text by removing punctuation (remembering where sentences start and end), case-folding (converting all characters to lower case), and tokenizing terms (breaking sentences into words). Then Swami removes stopwords (commonly used words that are unlikely to capture semantics, such as "to", "the", "be"), and stems the text to conflate word variants (e.g., "ran", "running", and "run") to improve term matching. Swami uses the Indri toolkit [262] for removing stopwords and stemming. Swami then tags the parts of speech using the Standford coreNLP toolkit [182].

4.3.2 Identifying Testable Specifications From the Documentation

As Section 4.2.1 described, Swami has two ways of deciding which sections of the specification to generate tests from. For many structured specifications such as ECMA-262, the sections that describe methods are clearly labeled with the name of the method (e.g., see Figure 4.1). For such specifications, Swami uses a regular expression it calls *Section Identification* to identify the relevant sections.

Swami's *Section Identification* regular expression discards white space and square brackets (indicating optional arguments), and looks for a numerical section label (which is labeled as "CD" by the parts of speech tagger), followed optionally by the "new" keyword (labeled "JJ"), a method name (labeled "NN"), then by a left parenthesis (labeled "LRB"), then arguments (labeled "NN"), and then a right parenthesis (labeled "RRB"). If the first line of a section matches this regular expression, Swami will attempt to generate tests; otherwise, this section is discarded. For example, the heading of the specification in Figure 4.1, "15.4.2.2 new Array (len)" is pre-processed to `[('15.4.2.2', 'CD'), ('new', 'JJ'), ('Array', 'NN'), ('(', '-LRB-'), ('len', 'NN'), (')', '-RRB-')]` matching the *Section Identification* regular expression.

If the specifications do not have a clear label, Swami can use information retrieval to identify which sections to generate tests for. (This step does not require the parts of speech tags.) The fundamental assumption underlying this approach is that some terms in a specification will be found in relevant source files. Swami considers the source code to be a collection of documents and the specification to be a query. Swami parses the abstract syntax trees of the source code files and identifies comments and class, method, and variable names. Swami then splits the extracted identifiers into tokens using CamelCase splitting. Swami will index both the split and intact identifies, as the specifications can contain either kind. Swami then tokenizes the specification and uses the Indri toolkit [262] for stopword removal, stemming, and indexing by computing the term frequency, the number of times a term occurs in a document, and document frequency, the number of documents in which a term appears. Finally, Swami uses the Indri toolkit to apply the Okapi information retrieval model [237] to map specification sections to source code, weighing more unique terms more strongly than more common terms. We chose to use the Okapi model because recent studies have found it to outperform more complex models on both text and source code artifacts [230, 267]. The Okapi model computes the inverse document frequency, which captures both the frequency of the co-occurring terms and the uniqueness of those terms [314]. This model also weighs more heavily class and method names, which can otherwise get lost in the relatively large number of variable names and comment terms. Swami will attempt to generate tests from all specifications with a similarity score to at least one class of 0.07 or above (recall the example in Figure 4.2). We selected this threshold by computing the similarity scores of a small subset (196 sections) of ECMA-262, selecting the minimum similarity score of the relevant specifications (where relevant is defined by the ground-truth [71], described in Section 4.5.4). This prioritizes recall over precision. More advanced models [236] might improve model

quality by improving term weights, but as Section 4.5.4 will show, this model produces sufficiently high-quality results.

4.3.3 Extracting Test Templates

To generate test templates, Swami uses rule-based natural language processing on the specification sections identified as relevant in Section 4.3.2. Swami applies four rules — *Template Initialization*, *Assignment Identification*, *Conditional Identification*, and *Conditional Translation* — in series. Each rule first uses a regular expression to decide if a sentence should be processed by the rule. If it should, the rule then applies other regular expressions to extract salient information, such as method names, argument names, assignments, conditional predicates, etc. When sentences do not match a rule, they are discarded from processing by further rules (recall Figure 4.4). Only the sentences that match all four rules will produce test templates.

Rule 1: ***Template Initialization*** locates and parses the name of the method being tested and the arguments used to run that method, and produces a test template with a valid method signature and an empty body. *Template Initialization* works similarly to the *Section Identification* regular expression; it matches the numerical section label, method name, and arguments by using their parts of speech tags and parentheses. Unlike *Section Identification*, *Template Initialization* also matches and extracts the method and argument names. Swami generates a test name by concatenating terms in the method under test's name with "_", and concatenating the extracted arguments with ", ", populating the following template:

```
1 function test_<func name> ([thisObj], <func args>){}
```

Swami uses the `thisObj` variable as an argument to specify `this`, an object on which this method will be invoked.

For example, the specification section "21.1.3.20 String.prototype.startsWith (searchString [,position])" results in the empty test template

```
1 function test_string_prototype_startswith(thisObj, searchString,
    position){}
```

Swami identifies the type of the object on which the tested method is invoked and stores the method's syntax in a database. For example, the `String.startsWith` method is invoked on *String* objects, so Swami stores `var output = new String(thisObj ).startsWith(searchString, position)` in the database.

Rule 2: ***Assignment Identification*** identifies assignment sentences and produces pairs of variables and their values, which it stores in Swami database. Specifications often use intermediate variables, described in the form of assignment sentences. For example, the specification of `ToUint32` (Figure 4.5) contained the assignment sentence "Let `posInt` be `sign(number)` × `floor(abs(number))`" and `posInt` was used later in the specification. Assignment sentences are of the form "Let `<var>` be `<value>`", where `<var>` and `<value>` are phrases. *Assignment Identification* uses a regular expression to check if a sentence satisfies this form. If it does, regular expressions extract the variable and value names, using the parts of speech tags and keywords, such as "Let". Values can be abstract operations (e.g., `ToUint32`), implicit operations (e.g., `max`), constant literals, or other variables. Each type requires a separate regular expression. For the example assignment sentence from Figure 4.5, Swami extracts variable `posInt` and value `sign(number)` × `floor(abs(number))`. Swami inserts this variable-value pair into its database. This populates the database with assigned variable-value pairs as specified by the specification.

Rule 3: ***Conditional Identification*** identifies conditional sentences that result in a return statement or the throwing of an exception, and produces source code (an `if-then` statement) that encodes the oracle capturing that behavior. Our key observation is that exceptional and boundary behavior is often specified in conditional sentences and that this behavior can be extracted by regular expressions, capturing both the predicate that should lead to the behavior and the expected behavior.

Conditional sentences are of the form "If <condition> <action>", where <condition> and <action> are phrases, often containing variables and method calls described in assignment statements. *Conditional Identification* uses three regular expressions to check if a sentence satisfies this form and if the <action> is a boundary condition or an exception. If `If * NN * is .* return .*` or `If * NN * the result is .*` match the sentence, Swami extracts the predicate <condition> as the text that occurs between the words `If` and either `return` or `the result is`. Swami extracts the <action> as the text that occurs after the words `return` or `the result is`. If `If * NN * is .* throw .* exception` matches the sentence, Swami extracts the predicate <condition> as the text that occurs between `If` and `throw` and the expected exception as the noun (NN) term between `throw` and `exception`. For example, consider the following three sentences taken from section 21.1.3.20 of the ECMA-262 specification for the `String.prototype.startsWith` method. The sentence "If isRegExp is true, throw a TypeError exception." results in a `condition-action` pair `isRegExp is true - TypeError`. The sentence "If searchLength+start is greater than len, return false." results in `searchLength+start is greater than len - false`. And the sentence "If the sequence of elements of S starting at start of length searchLength is the same as the full element sequence of searchStr, return true." results in `the sequence of elements of S starting at start of length searchLength is same as the full element sequence of searchStr - true`.

Finally, *Conditional Identification* generates source code by filling in the following templates for exceptional behavior:

```
1 if (<condition>){
2     try{
3         <function call>
4         return
5     }catch(e){
6         <test constructor>(true, eval(e instanceof <action>))
7         return
8     }
9 }
```

and for `return` statement behavior:

```
1 if (<condition>){
2     <function call>
3     <test constructor>(output, <action>))
4     return
5 }
```

The `<function call>` placeholder is replaced with the method invocation code that is generated by *Template Initialization* (and stored in the database) to invoke the method under test, e.g., `var output = new String(thisObj).startsWith(searchString, position)`. The `<test constructor>` placeholder is a project-specific description of how test cases are written, and is an input to Swami (recall Figure 4.4 and Section 4.2); for Rhino, it is `new TestCase(test name, test description, expected output, actual output)`. Figure 4.6 shows the code *Template Initialization* generates from the three conditional sentences in the `String.startsWith` specification.

Rule 4: *Conditional Translation* processes the oracle code generated by *Conditional Identification* by recursively filling in the variable value assignments according to the specification and injects the code into the template produced by *Template Initialization*, producing a complete test template.

Conditional Translation identifies the intermediate variables in the code generated by *Conditional Identification* and replaces them with their values from the database (placed in the database by *Assignment Identification*). If an intermediate variable does not have a value in the database (e.g., because the specification was incomplete or because the text describing this variable did not match *Assignment Identification*'s regular expressions), this test will fail to compile and Swami will remove the entire condition from the test. Next, Swami translates the implicit operations in the code. For example, `is greater than or equal to` and $\geq$ are translated to `>=`; `this value` is translated to `thisObj`; `is exactly`, `is equal to`, and `is` are translated to `===`; `<x> is one of <a>, <b>, ...` is translated to `x===a || x===b...`; `the number of elements in <S>` is

```
1  if (isRegExp is true){
2      try{
3          var output = new String(thisObj).startsWith(searchString,
               position)
4          return
5      }catch(e){
6          new TestCase(<test name>, <test description>, true, eval(e
               instanceof TypeError))
7          return
8      }
9  }
10 if (searchLength+start is greater than len){
11     var output = new String(thisObj).startsWith(searchString,
           position)
12     new TestCase(<test name>, <test description>, output, false))
13     return
14 }
15 if (the sequence of elements of S starting at start of length
       searchLength is same as the full element sequence of searchStr)
       {
16     var output = new String(thisObj).startsWith(searchString,
           position)
17     new TestCase(<test name>, <test description>, output, true))
18     return
19 }
```

Figure 4.6: The test code generated by the *Template Initialization* rule from the JavaScript `String.startsWith` specification.

translated to `<S>.length`; etc. Swami contains 54 such patterns, composed of keywords, special characters, wildcard characters, and parts of speech tags.

An inherent limitation of the regular expressions is that they are rigid, and fail on some sentences. For example, Swami fails on the sentence "If the sequence of elements of S starting at start of length searchLength is the same as the full element sequence of searchStr, return true." Swami correctly encodes most variable values and implicit operations, but fails to encode `the sequence of elements of S starting at the start of length searchLength`, resulting in a non-compiling `if` statement, which Swami removes as a final post-processing step.

```
1  function test_string_prototype_startswith(thisObj, searchString,
       position){
2      if (isRegExp === true){
3          try{
4              var output = new String(thisObj).startsWith(
                   searchString, position)
5              return
6          }catch(e){
7              new TestCase(<test name>, <test description>, true,
                   eval(e instanceof TypeError))
8              return
9          }
10     }
11     if (ToString(searchString).length + Math.min(Math.max(
           ToInteger(position), 0), ToString(RequireObjectCoercible(
           thisObj)).length) > ToString(RequireObjectCoercible(thisObj
           )).length)){
12         var output = new String(thisObj).startsWith(searchString,
               position)
13         new TestCase(<test name>, <test description>, output,
               false))
14         return
15     }
16 }
```

Figure 4.7: The final test template Swami generates for the `String.startsWith` JavaScript method.

Swami adds each of the translated conditionals that compiles to the test template initialized by *Template Initialization*. Figure 4.7 shows the final test template Swami generates for the `String.startsWith` JavaScript method.

4.3.4 Generating Executable Tests

Swami instantiates the test template via heuristic-driven random input generation. ECMA-262 describes five primitive data types: Boolean, Null, Undefined, Number, and String, and two non-primitive data types, Symbol and Object. Swami uses the five primitive data types and several subtypes of Object (Array, Map, Math, DateTime, RegExp, etc.). For Boolean, Swami generates `true` and `false`; for Null, `null`; for Undefined, `undefined`; for Number, random integers and floating point values. Swami also uses the following special values: `NaN`, `-0`, `+0`, `Infinity`, and `-Infinity`. For the

Object subtypes, Swami follows several heuristics for generating Maps, Arrays, regular expressions, etc. For example, when generating an Array, Swami ensures to generate only valid length arguments (an integer between 1 and 2^{32}). (Invalid lengths throw a `RangeError` at runtime, which would result in false alarm test failures.)

Finally, Swami augments the test suite with the developer written implementation of abstract operations used in the specification. Recall that for JavaScript, this consisted of 82 total lines of code, and that most of these abstract operation implementations can be reused for other specifications.

4.4 Subjects of Investigation

We evaluate Swami using ECMA-262, the official specification of the JavaScript programming language [290], and two well known JavaScript implementations: Java Rhino and C++ Node.js built on Chrome's V8 JavaScript engine. We chose ECMA-262 because: (1) it is more reliable and regularly updated unlike JavaDoc comments, which developers forget to update while updating their source code, and (2) multiple independently-maintained, open-source JavaScript implementations adhere to ECMA-262 specifications, which gives us a good dataset to evaluate the effectiveness of Swami-generated tests.

4.5 Evaluating Swami-Generated Tests and Key Findings

In this section, we first summarize the key findings of this study in terms of the research questions we ask. We then describe the detailed results that lead to the summarized findings.

We evaluate Swami's performance to answer the following four research questions.

RQ1 How precise are Swami-generated tests?

Answer: Of the tests Swami generates, 60.3% are innocuous — they can never fail. Of the remaining tests, 98.4% are precise to the specification and only 1.6% are flawed and might raise false alarms.

RQ2 Do Swami-generated tests cover behavior missed by developer-written tests?

Answer: Swami-generated tests identified 1 previously unknown defect and 15 missing JavaScript features in Rhino, 1 previously unknown defect in Node.js, and 18 semantic ambiguities in the ECMA-262 specification. Further, Swami generated tests for behavior uncovered by developer-written tests for 12 Rhino methods. The average statement coverage for these methods improved by 15.2% and the average branch coverage improved by 19.3%.

RQ3 Do Swami-generated tests cover behavior missed by state-of-the-art automated test generation tools?

Answer: We compare Swami to EvoSuite and find that most of the EvoSuite-generated failing tests that cover exceptional behavior are false alarms, whereas 98.4% of the Swami-generated tests are precise to the specification and can only result in true alarms. Augmenting EvoSuite-generated tests using Swami increased the statement coverage of 47 classes by, on average, 19.5%. Swami also produced fewer false alarms than Toradacu and Jdoctor, and, unlike those tools, generated tests for missing features.

RQ4 Does Swami's Okapi model precisely identify relevant specifications?

Answer: The Okapi model's precision is 79.0% and recall is 98.9%, but further regular expressions can remove improperly identified specifications, increasing precision

to 93.1%; all the tests generated from the remaining improperly identified specifications fail to compile and can be automatically discarded.

In the following sections, we describe the details about the results obtained for each research question.

4.5.1 Evaluating Precision of Swami-Generated Tests

RQ1: How precise are Swami-generated tests?

Swami's goal is generating tests for exceptional behavior and boundary conditions, so we do not expect it to produce tests for most of the specifications. Instead, in the ideal, Swami produces highly-precise tests for a subset of the specifications.

We use Swami to generate black-box tests from ECMA-262 (v8) JavaScript specifications. Swami generated test templates for 98 methods, but 15 of these test templates failed to compile, resulting in 83 compiling templates. We manually examined the 83 test templates and compared them with the natural language specifications to ensure that they correctly capture specification's oracles. We then instantiated each template with 1,000 randomly generated test inputs, creating 83,000 tests. We instrumented test templates to help us classify the 83,000 tests into three categories: (1) good tests that correctly encode the specification and would catch some improper implementation, (2) bad tests that incorrectly encode the specification and could fail on a correct implementation, and (3) innocuous tests that pass on all implementations.

Of the 83,000 tests, 32,379 (39.0%) tests were good tests, 535 (0.6%) were bad tests, and 50,086 (60.3%) were innocuous.

It is unsurprising that innocuous tests are common. For example, if a test template checks that a RangeError exception is thrown (recall Figure 4.1), but the randomly-generated inputs are not outside the allowed range, the test can never fail. Innocuous

tests cannot raise false alarms, since they can never fail, but could waste test-running resources. Existing test prioritization tools may be able to remove innocuous tests.

We manually analyzed the 535 bad tests. (Of course, many of these tests could be grouped into a few equivalence classes based on the behavior they were triggering.) All bad tests came from 3 specifications. First, 176 of the tests tested the `ArrayBuffer.prototype.slice` method, and all invoked the method on objects of type `String` (instead of `ArrayBuffer`). Because `slice` is a valid method for `String` objects, that other method was dynamically dispatched at runtime and no `TypeError` exception was thrown, violating the specification and causing the tests to fail.

Second, 26 of the bad tests tested the `Array.from(items,mapfn,thisArg)` method, expecting a `TypeError` whenever `mapfn` was not a function that can be called. The 26 tests invoke the `Array.from` with `mapfn` set to `undefined`. While `undefined` cannot be called, the specification describes different behavior for this value, but Swami fails to encode that behavior.

Third, 333 of the tests tested the `Array(len)` constructor with nonnumeric `len` values, such as Strings, booleans, etc. This dynamically dispatched a different `Array` constructor based on the type of the argument, and thus, no `RangeError` was thrown.

These imprecisions in Swami's tests illustrate the complexity of disambiguating natural language descriptions of code elements. The tests were generated to test specific methods, but different methods executed at runtime. Swami could be improved to avoid generating these bad tests by adding more heuristics to the input generation algorithm.

Of the non-innocuous tests Swami generates, 98.4% are precise to the specification and only 1.6% are flawed and might raise false alarms.

4.5.2 Comparing Swami Tests With the Developer Tests

RQ2: Do Swami-generated tests cover behavior missed by developer-written tests?

Rhino and Node.js are two extremely well-known and mature JavaScript engines. Rhino, developed by Mozilla, and Node.js, developed by the Node.js Foundation on Chrome's V8 JavaScript engine, both follow rigorous development processes that include creating high-quality regression test suites. We compared the tests Swami generated to these developer-written test suites for Rhino v1.7.8 and Node.js v10.7.0 to measure if Swami effectively covered behavior undertested by developers.

Rhino's developer-written test suites have an overall statement coverage of 71% and branch coverage of 66%. Swami generated tests for behavior uncovered by developer-written tests for 12 Rhino methods, increasing those methods' statement coverage, on average, by 15.2% and branch coverage by 19.3%. For Node.js, the developer-written test suites are already of high coverage, and Swami did not increase Node.js statement and branch coverage. However, coverage is an underestimate of test suite quality [276], as evidenced by the fact that Swami discovered defects in both projects.

Swami generated tests that identified 1 previously unknown defect and 15 missing JavaScript features in Rhino, 1 previously unknown defect in Node.js, and 18 semantic ambiguities in the ECMA-262 specification. The Rhino issue tracker contains 2 open feature requests (but no tests) corresponding to 12 of 15 missing features in Rhino. We have submitted a bug report for the new defect[1] and a missing feature request for the 3 features not covered by existing requests[2].

The discovered defect dealt with accessing the `ArrayBuffer.prototype.byteLength` property of an `ArrayBuffer` using an object other than an `ArrayBuffer`. Neither Rhino nor Node.js threw a `TypeError`, failing to satisfy the JavaScript specification.

---

[1] `https://github.com/mozilla/rhino/issues/522`

[2] `https://github.com/mozilla/rhino/issues/521`

The missing features dealt with 6 `Map` methods[3], 6 `Math` methods[4], 2 `String` methods (`padStart` and `padEnd`), and 1 `Array` method (`includes`) not being implemented in Rhino, failing to satisfy the JavaScript specification.

The 18 semantic ambiguities are caused by JavaScipt's multiple ways of comparing the equality of values, e.g., `==`, `===`, and `Object.is`. In particular, `===` and `Object.is` differ only in their treatment of `-0`, `+0`, and `NaN`. The specification often says two values should be equal without specifying which equality operator should be used. This causes a semantic ambiguity in the specifications of methods that differentiate between these values. In fact, developer-written tests for Node.js and Rhino differ in which operators they use to compare these values, with Rhino's developers implementing their own comparator and explicitly stating that they consider differentiating between `+0` and `-0` unimportant[5], whereas the specification dictates otherwise. With this ambiguity in the specification, Swami has no obvious way to infer which equality operator should be used and can generate imprecise tests. When using `===`, Swami generated 18 false alarms revealing this ambiguity. Using `Object.is` removes these false alarms.

Swami-generated tests identified 1 previously unknown defect and 15 missing JavaScript features in Rhino, 1 previously unknown defect in Node.js, and 18 semantic ambiguities in the ECMA-262 specification. Further, Swami generated tests for behavior uncovered by developer-written tests for 12 Rhino methods. The average statement coverage for these methods improved by 15.2% and the average branch coverage improved by 19.3%.

---

[3] `https://github.com/mozilla/rhino/issues/159`

[4] `https://github.com/mozilla/rhino/issues/200`

[5] `https://github.com/mozilla/rhino/blob/22e2f5eb313b/testsrc/tests/shell.js`

4.5.3 Comparing Swami Tests With the State-Of-The-Art Test Generation Tool

RQ3: Do Swami-generated tests cover behavior missed by state-of-the-art automated test generation tools?

Random test generation tools, such as EvoSuite [81] and Randoop [211], can generate tests in two ways: using explicit assertions in the code (typically written manually), or as regression tests, ensuring the tested behavior doesn't change as the software evolves. As such, Swami is fundamentally different, extracting oracles from specifications to capture the intent encoded in those specifications. Still, we wanted to compare the tests generated by Swami and EvoSuite, as we anticipated they would cover complementary behavior.

We used EvoSuite to generate five independent test-suites for Rhino using 5 minute time budgets and line coverage as the testing criterion, resulting in 16,760 tests generated for 251 classes implemented in Rhino. Of these, 392 (2.3%) tests failed because the generated inputs resulted in exceptions EvoSuite had no oracles for (Rhino source code did not encode explicit assertions for this behavior). This finding is consistent with prior studies of automated test generation of exceptional behavior [22]. By comparison, only 1.6% of the non-innocuous Swami-generated tests and only 0.6% of all the tests were false alarms (recall RQ1). Swami significantly outperforms EvoSuite because it extracts oracles from the specifications.

Since EvoSuite, unlike Swami, requires project source code to generate tests, EvoSuite failed to generate tests for the 15 methods that exposed missing functionality defects in Rhino, which Swami detected by generating specification-based tests.

The EvoSuite-generated test suite achieved, on average, 77.7% statement coverage on the Rhino classes. Augmenting that test suite with Swami-generated tests increased the statement coverage of 47 classes by 19.5%, on average.

The tools most similar to Swami, Toradacu [90] and Jdoctor [22], are difficult to compare to directly because they work on Javadoc specifications and cannot generalize to the more complex natural language specifications Swami can handle. On nine classes from Google Guava, Toradocu reduced EvoSuite's false alarm rate by 11%: out of 290 tests, the false alarms went from 97 (33%) to 65 (22%) [90]. Meanwhile, on six Java libraries, Jdoctor eliminated only 3 of Randoop's false alarms out of 43,791 tests (but did generate 15 more tests and correct 20 tests' oracles) [22]. And again, without an implementation (or without the associated Javadoc comments), neither Toradacu nor Jdoctor can generate tests to identify the missing functionality Swami discovered. Overall, Swami showed more significant improvements in the generated tests.

> We compare Swami to EvoSuite and find that most of the EvoSuite-generated failing tests that cover exceptional behavior are false alarms, whereas 98.4% of the Swami-generated tests are precise to the specification and can only result in true alarms. Augmenting EvoSuite-generated tests using Swami increased the statement coverage of 47 classes by, on average, 19.5%. Swami also produced fewer false alarms than Toradacu and Jdoctor, and, unlike those tools, generated tests for missing features.

4.5.4 Evaluating Swami's Precision to Identify Relevant Specifications

RQ4: Does Swami's Okapi model precisely identify relevant specifications?

Swami's regular expression approach to *Section Identification* is precise: in our evaluation, 100% of the specification sections identified encoded testable behavior. But it requires specific specification structure. Without that structure, Swami relies on its Okapi-model approach. We now evaluate the Okapi model's precision (the fraction of the sections the model identifies as relevant that are actually relevant) and recall (the fraction of all the relevant sections that the model identifies as relevant).

Eaddy et al. [70] have constructed a ground-truth benchmark by manually mapping parts of the Rhino source code (v1.5R6) to the relevant concerns from ECMA-262 (v3). Research on information retrieval in software engineering uses this benchmark extensively [71, 100]. The benchmark consists of 480 specifications and 140 Rhino classes. On this benchmark, Swami's precision was 79.0% and recall was 98.9%, suggesting that Swami will generate tests from nearly all relevant specifications, and that 21.0% of the specifications Swami may consider generating tests from may not be relevant.

However, of the irrelevant specifications, 45.3% do not satisfy the *Template Initialization* rule, and 27.3% do not satisfy the *Conditional Identification* rule. All the test templates generated from the remaining 27.4% fail to compile, so Swami removes them, resulting in an effective precision of 100%.

> The Okapi model's precision is 79.0% and recall is 98.9%, but further regular expressions can remove improperly identified specifications, increasing precision to 93.1%; all tests generated from the remaining improperly identified specifications fail to compile and can be automatically discarded.

4.6 Comparing Swami Approach With the State-Of-The-Art

Our research complements prior work on automatically generating test inputs for regression tests or manually-written oracles, such as EvoSuite [81] and Randoop [211], by automatically extracting oracles from natural language specifications. The closest work to ours is Jdoctor [22] and Toradacu [90], which extract oracles for exceptional behavior, and @tComment [265], which focuses on extracting preconditions related to nullness of parameters. These techniques are limited to using Javadoc comments, which are simpler than the specifications Swami tackles because Javadoc comments (1) provide specific annotations for pre- and post-conditions, including `@param`,

`@throws`, and `@returns`, making them more formal [265]; (2) are collocated with the method implementations they specify, (3) use the variable names as they appear in the code, and (4) do not contain references to abstract operations specified elsewhere. Additionally, recent work showed that Javadoc comments are often out of date because developers forget to update them when requirements change [265]. Our work builds on @tComment, Toradacu, and Jdoctor, expanding the rule-based natural language processing techniques to apply to more complex and more natural language. Additionally, unlike those techniques, Swami can generate oracles for not only exceptional behavior but also boundary conditions. Finally, prior test generation work [22, 81, 90, 211] requires access to the source code to be tested, whereas Swami can generate black-box tests entirely from the specification document, without needing the source code.

4.7 Discussion

While Swami's regular-expression-based approach is rather rigid, it performs remarkably well in practice for exceptional and boundary behavior. It forms both a useful tool for generating tests for such behavior, and a baseline for further research into improving automated oracle extraction from natural language by using more advanced information retrieval and natural language processing techniques. Augmenting developer-written test suite and automatically-generated test suite with Swami tests significantly improved both kinds of test suites by covering code related to under-tested functionality. Thus, Swami can be useful to improve APR quality by enabling APR techniques to produce better patches using the improved developer-written test suites, and by filtering out the plausible patches by using better held-out, automatically-generated evaluation test suites.

4.8 Threats to Validity

Regular expressions are brittle and may not generalize to other specifications or other software. We address this threat in three ways. First, our evaluation uses ECMA-based specifications which has been used to specify hundreds of systems [12], from software systems, to programming languages, to Windows APIs, to data communication protocols, to telecommunication networks, to data storage formats, to wireless proximity systems, and so on. This makes our work directly applicable to at least that large number of systems, and it can likely can be extended to other standards. Second, our evaluation focuses on the specification of JavaScript, a mature, popular language. The ECMA-262 standard is the official specification of JavaScript. Third, we evaluate our approach on two well-known, mature, well-developed software systems, Rhino, written in Java, and Node.js, in C++, demonstrating generalizability to test generation for different languages and systems.

The correctness of Swami-generated tests relies on the test constructor written by the developer. For example, the Rhino's test case constructor internally uses method `getTestCaseResult(expected, actual)` that fails to distinguish `-0` from `+0` (recall Section 4.5.2). This inhibits the Swami-generated tests from correctly testing methods whose behavior depends on differentiating signed zeros.

Our evaluation of the Okapi-model-based approach and our choice of similarity score threshold rely on a manually-created ground truth benchmark [70]. Errors in the benchmark may impact our evaluation. We mitigate this threat by using a well-established benchmark from prior studies [71, 100, 101].

4.9 Contributions

We have presented Swami, a regular-expression-based approach to automatically generate oracles and tests for boundary and exceptional behavior from structured natural language specifications. Swami is the first approach to work on specifications

as complex as the ECMA-262 JavaScript standard. Our evaluation demonstrates that Swami is effective at generating tests, complements tests written by developers and generated by EvoSuite, and finds previously unknown bugs in mature, well-developed software. The main contributions of this work are:

- Swami, an approach for generating tests from structured natural language specifications.
- An open-source prototype Swami implementation, including rules for specification documents written in ECMA-script style, and the implementations of common abstract operations.
- An evaluation of Swami on the ECMA-262 JavaScript language specification, comparing Swami-generated tests to those written by developers and those automatically generated by EvoSuite, demonstrating that Swami generates tests often missed by developers and other tools and that lead to discovering several unknown defects in Rhino and Node.js.
- A replication package of all the artifacts and experiments described in chapter available at `http://swami.cs.umass.edu/`.

This work is joint with Yuriy Brun, and credit for this work is shared between the two of us. The published version of this study [197] can be found at `https://doi.org/10.1109/ICSE.2019.00035`.

CHAPTER 5

IMPROVING FAULT LOCALIZATION USING BUG REPORTS

5.1 Introduction

Identifying the defective program elements (e.g., classes, methods, or statements) is the first step in repairing software defects whether manually or automatically. Fault localization (FL) research focuses on automatically identifying defective program elements that cause software failures. Most of the automated FL techniques use dynamic analysis and run-time information of the defective program to compute the suspiciousness score (probability of being defective) of the program elements. The developer or an APR tool can then use a ranked list of program elements to fix the defect. Please refer to the survey study [293] for more details on the state-of-the-art FL techniques. Studies show that accuracy of the FL used by APR has a significant effect on APR's success [5, 11, 110, 164, 286, 304], and manually improving FL can correctly patch more defects [5, 166]. Some APR tools, such as SimFix [111] use specific heuristics to address inaccurate FL, however, their FL is tightly coupled with their repair tool implementation making it non-reusable for other repair tools. This problem is known as FL bias in APR [164, 166]. The goal of the study described in this chapter is to develop a reusable and better FL for APR by combining information from test executions and bug reports.

Depending on the source of information used to localize the fault, FL techniques can be classified into multiple families. For example, spectrum-based (SBFL) techniques (e.g., [2, 99, 295]) use test coverage information, mutation-based (MBFL) techniques

(e.g., [196, 213]) use test results collected from mutating the program, (dynamic) program slicing techniques (e.g., [6,234]) use the dynamic program dependencies, stack trace analysis techniques (e.g., [291, 294]) use error messages, predicate switching techniques (e.g., [317]) use test results from mutating the results of conditional expressions, information retrieval-based FL (IRFL) techniques (e.g., [319], [243]) use bug report information, and history-based FL (e.g., [124, 229]) use the development history to identify the suspicious program elements that are likely to be defective. Recent studies in FL have found that none of the families of FL techniques is the best and combining multiple techniques across different families outperforms individual techniques [155, 320].

Even though there exists a variety of FL techniques, most APR tools typically use SBFL techniques that uses test execution coverage to compute the suspiciousness scores of program elements. While researchers are actively working on improving FL by using multiple bug information sources, to the best of our knowledge, there does not exist any repair tool that uses these advancements to localize bugs in the repair process. The work closest to experimenting with FL in APR is a repair technique iFixR [132] that internally uses bug reports and an IRFL FL technique to localize bugs. iFixR patches defects that 16 SBFL-based repair tools cannot, and vice versa. In this chapter, we describe FL techniques that use bug reports to localize defects and are suitable for use in APR. We describe (1) a light-weight FL technique that uses structured information-retrieval-based approach to localize defects using bug reports at the statement-level and (2) an unsupervised-learning-based technique that uses rank aggregation algorithms to combine multiple FL techniques, which may use different bug information sources. While existing research in FL focuses on improving manual program repair, the FL techniques described in this chapter intend to improve program repair.

This chapter is organized as follows. Section 5.2 describes Blues, a technique that localizes bugs by ranking suspicious program statement using bug reports. Section 5.3 describes our implementation of an SBFL technique, which is popularly used by most test-suite-based program repair tools. Section 5.4 describes our method for combining FL techniques called RAFL and using RAFL to combine Blues and our SBFL into SBIR. Section 5.5 describes the real-world defects and evaluation metrics used to evaluate the performance of FL techniques. Section 5.6 describes the evaluation and key findings in terms of the research questions. Section 5.7 discusses the use of FL techniques in program repair. Section 5.8 addresses the threats to validity of our study and Section 5.9 summarizes our contributions.

5.2 Blues: Localizing Bugs Using Bug Reports

We create Blues because most existing IRFL techniques [131, 243, 288, 291, 310, 319] are not well suited for APR as they localize defects to a file or a method, whereas APR tools need finer, statement-level localization. The only known statement-level IRFL is used internally by iFixR [132], but is poorly suited for our needs because (1) it uses *supervised* learning, (unlike Blues) requiring a dataset with bug reports, results from six IRFL techniques, and correct defect locations, and (2) it is pre-trained on projects [153] that overlap with the Defects4J benchmark used in our evaluations and re-training D&C poses complex technical challenges. Further, iFixR's FL technique ignores localizing defects in several abstract syntax tree (AST) statements including for and while loops that can improve APR [163], which Blues uses.

Our IRFL technique, Blues, uses bug reports to localize defects at the statement level. Blues does not reinvent the wheel. It starts with BLUiR [243] to identify suspicious files (Section 5.2.1), and then extracts data on 57 types of source-code statements to extend BLUiR's mechanism to the statement level and develops several methods for combining the file-level and statement-level results (Section 5.2.2).

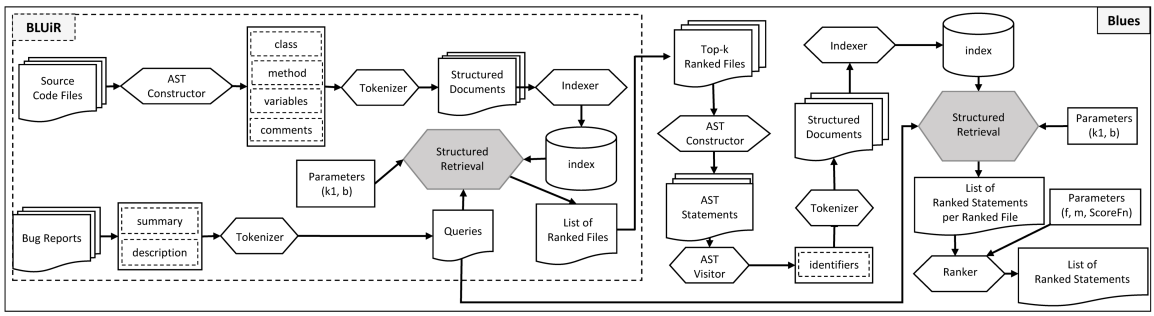


Figure 5.1: The Blues architecture. The Blues architecture builds on BLUiR [243] to rank suspiciousness program statements using structured information retrieval.

5.2.1 Ranking Suspicious Files

Blues builds on BLUiR [243], an existing file-level IRFL technique. BLUiR uses structured information retrieval to compute the similarity between bug reports and source code files. We select BLUiR because it is efficient, does not require a training dataset, and performs comparably to the other state-of-the-art file-level IRFL techniques [153].

Figure 5.1 shows the Blues architecture. Blues extends BLUiR to produce statement-level results. For each defect, Blues' inputs are the source files and the bug report. Blues builds the AST of each source file using Eclipse Java Development Tools (JDT). Blues processes the AST of source files to extract identifiers associated with each program construct, such as class names, method names, variable names, and comments. It then splits the extracted identifiers into tokens using *CamelCase* splitting, which improves the matching recall. Blues then parses the bug report to extract identifiers from the *summary* and *description* fields, storing the information in separate structured XML documents. The XML documents created from source files and bug report are then fed into Indri toolkit [262] for efficient indexing and for developing the retrieval model. Indri pre-processes the XML documents using text normalization (remove punctuation, perform case-folding, tokenize terms), stopword removal (remove extraneous terms such as "a", "the", "be", etc.), and stemming

(conflate variants of the same underlying term e.g., “ran”, “running”, “run”). Next, it indexes the pre-processed documents by collecting and storing statistics, such as term frequency (TF) (the number of times a term occurs in a given document), document frequency (DF) (the number of documents in which a given term appears), and Inverse Document Frequency (IDF), which is formulated as $log(\frac{N}{DF})$, where N is the total number of documents in the collection. Finally, Blues uses an IR model (TF-IDF formulation based on the BM25 (Okapi) model [238]) to search and rank the documents based on their similarity with the given bug report. The TF-IDF-based IR model uses two tuning parameters: the term weight scaling parameter k_1 and the document normalization parameter b, which are provided as input (along with XML documents) to the Indri toolkit. We set $k_1 = 1.0$ and $b = 0.3$, as suggested in the original BLUiR study [243] tuned using independent dataset. The output of the IR model is the ranked list of source files along with similarity scores for the bug report.

5.2.2 Ranking Suspicious Statements

To identify the suspicious statements from the top-ranked suspicious files, Blues takes the top-k ranked files, uses Eclipse JDT to create the AST of each source file, and then uses AST Visitor to parse and extract AST statements from the source file ASTs. Prior work [163] shows that localizing bugs at the expression-level can improve APR tools. Therefore, unlike the IRFL technique used in iFixR [132], which only extracts five kinds of AST statements (if, return, FieldDeclaration, Expression, and VariableDeclaration), Blues extracts 32 AST expressions [231], 3 AST nodes (Annotation, SingleVariableDeclaration, AnonymousClassDeclaration), and 22 AST statements [73], 17 of which iFixR ignores, including for loops, while loops, do statements, array access, method invocation, and so on). For readability, we refer to the AST node, AST expression, and AST statement as *statement*.

For each statement, Blues extracts the identifier terms and the line number of that statement. Extracting AST statements instead of natural language text from source files enables Blues to (1) fetch compilable code statements (a single code statement may span across multiple lines in the source file), which can be considered for replacement by repair tools, and (2) compute separate suspiciousness scores for nested statements that exist on the same line of the source file. Next, for each statement, Blues creates an XML document that contains the identifiers extracted from that statement along with the information of its source file and line number. Blues then feeds these XML documents, along with the same tuning parameters ($k_1 = 1.0$ and $b = 0.3$) used for ranking source files, to the Indri toolkit to perform the same processing as for ranking source files, and uses the same IR model to produce a ranked list of statements along with their similarity scores with the bug report. Blues extracts the source file and line number information from ranked statement results to produce a ranked list of suspicious statements per file.

Real-world projects contain many source files, and it may be ineffective to consider all statements in a higher-ranked file to be more suspicious ones in lower-ranked files. While sometimes effective, our experiments show that this strategy is sometimes sub-optimal, so we also explore other strategies. Blues provides a ranker module that uses three parameters to produce a final ranked list:

f**:** the number of suspicious files to consider.

m**:** the number of suspicious statements per file to consider.

*ScoreFn***:** a function for combining the file and statement suspiciousness scores. We define two such functions:

1. $Score_{high}$ ranks the m most suspicious statements in the most suspicious file, followed by m statements in the next file, and so on.

2. $Score_{wt}$ uses the files' scores as weights for the associated suspicious statements and recomputes the weighted suspiciousness scores for the statements by multiplying the suspiciousness scores of the statements with the suspiciousness score of the associated file. The statements are then ranked using the weighted scores. This is same as the method used by iFixR [132] to combine file and statement scores.

In our experiments, we use $f = 50$ based on the recommendation of a prior study [132]. We experiment with using different m values and scoring functions. We run Blues' ranker module using six different configurations: five ($m \in \{1, 25, 50, 100, all\}$) with $Score_{high}$, and one ($m = all$) with $Score_{wt}$. For each of the six configurations, Blues produces a ranked list of suspicious statements.

Figure 5.2 shows the number of defects localized in top-k ($k \in \{1, 25, 50, 100,$ all$\}$) ranked lists obtained using the six configurations. The six configurations localize complementary defects, as evident by the row showing the union of defects localized. Therefore, we use Algorithm 1 to combine the six ranked lists into a single list, which we call *Blues ensemble*. The intuition behind our algorithm is that a statement with a higher suspiciousness score, considering all the six lists, be placed higher in the combined list. We break ties based on the number of lists in which statements occur; a statement found in more lists is ranked higher in the combined list. To fairly compare suspiciousness scores across lists, the algorithm uses normalized scores. As shown in Figure 5.2, for all list sizes we consider, Blues ensemble consistently localizes more defects than all of the underlying six configurations. Note that computing the individual configurations and ensemble is a relatively low-cost process. One only needs to rerun Blues's ranker module and Algorithm 1, not the entire Blues pipeline. From here on, we use only the ensemble and refer to it as just Blues.

m	ScoreFn	$k = 1$	25	50	100	all
1	$Score_{high}$	26	62	65	65	65
25	$Score_{high}$	26	153	219	269	384
50	$Score_{high}$	26	153	195	279	476
100	$Score_{high}$	26	153	195	241	526
all	$Score_{high}$	26	153	195	241	593
all	$Score_{wt}$	25	176	228	298	593
	union	30	224	300	377	593
Blues ensemble		**26**	**183**	**238**	**304**	**593**

Figure 5.2: Comparing the performance of Blues ensemble with underlying Blues configurations to localize 815 defects in Defects4J (v2.0). The six Blues configurations localize complementary defects (evident by "union") and Blues ensemble consistently localizes more defects than all of the underlying six configurations.

5.3 Spectrum-Based Fault Localization

A program spectrum is a measurement of the runtime behavior of a program, such as code coverage of developer-written tests [99]. Comparing program spectra on passing and failing tests can be used to rank program elements (e.g., class, method, statement). SBFL techniques calculate the suspiciousness score of an element using some ranking strategy that considers the following four values collected from the test execution coverage on that element: (1) number of failing tests that execute element (e_f), (2) number of failing tests that do not execute element (n_f), (3) number of passing tests that execute element (e_p), and (4) number of passing tests that do not execute element (n_p). While there are multiple ranking strategies proposed for SBFL, including Ochiai [2], DStar [292], and Tarantula [116], many empirical studies [301,320] have shown that Ochiai is more effective for object-oriented programs. Thus, most SBFL-based repair tools use Ochiai, and so does our study.

We do not create a new SBFL technique, but combine existing tools to produce a state-of-the-art implementation. There exist multiple frameworks that APR tools use to compute code coverage while implementing their SBFL techniques, including

Algorithm 1 Combining ranked lists using maximum suspiciousness scores and breaking ties using consensus.

```
Input: rankedLists ← [l_1, l_2, ...]  ▷ set of individual ranked lists to be combined
Output: combinedList                  ▷ combined list of ranked suspicious statements
procedure CombineLists(rankedLists)
    stmt_maxscore ← {}    ▷ stores max suspiciousness score of a stmt in all lists
    stmt_listcount ← {}   ▷ stores number of lists in which a stmt occurs
    combinedList ← [ ]    ▷ stores output combined list of stmts with scores
    for l_k ∈ rankedLists do
        list_n ← NormalizeScores(l_k)   ▷ normalize stmt scores to range in [0, 1]
        for (stmt, score) ∈ list_n do
            if score > 0.0 then
                if stmt ∉ stmt_maxscore then   ▷ stmt is seen for the first time
                    stmt_maxscore[stmt] = score
                    stmt_count[stmt] = 1
                else   ▷ if stmt is seen before, check if new score is higher
                    if score > stmt_maxscore[stmt] then
                        stmt_maxscore[stmt] = score   ▷ update max score if higher
                    stmt_count[stmt] = stmt_count[stmt] + 1   ▷ update count
    combinedList ← sortUsingScoreAndCount(stmt_maxscore, stmt_count)
▷ sort stmts based on max scores and break ties by placing stmts with higher count
higher in the list
    return combinedList
```

JaCoCo [103], GZoltar [34], and Cobertura [43]. Our study uses GZoltar because most APR tools use it, and a recent study comparing 14 APR tools used multiple GZoltar versions, showing that the latest-at-the-time version (v1.6.0) significantly improved FL results and repair performance [164]. We use the latest version (v1.7.2) of GZoltar. GZoltar's inputs are the source code and test suite and its outputs are each statement's e_f, n_f, e_p, and n_p. We use the the Ochiai ranking formula to compute suspiciousness scores:

$$score = \frac{e_f}{\sqrt{(e_f+n_f)(e_f+e_p)}}$$

We validate our SBFL implementation by comparing it to previously reported results [164] on Defects4J (v1.2.0) for SBFL implemented using Ochiai and older versions of Gzoltar. Figure 5.3 shows our SBFL implementation localizes 23 more defects than the best prior version.

project	Closure	Lang	Math	Mockito	Time	Total
#defects	133	65	106	38	26	368
GZ v0.1.1	78	29	91	21	22	241
GZ v1.6.0	95	57	100	23	22	297
GZ v1.7.2	**101**	**53**	**95**	**36**	**21**	**306**

Figure 5.3: Comparing FL performance of our SBFL with prior implementations. Our SBFL (implemented using GZoltar (v1.7.2) and Ochiai ranking strategy), in **bold**, localizes more defects than prior SBFLs using older versions of Gzoltar [164].

In the remainder of this chapter, when we refer to our SBFL, we are referring to this particular implementation.

5.4 Combining Fault Localization Techniques

Existing approaches to combine multiple FL techniques [140, 155, 255, 301, 320], are typically based on *learning to rank* [32] supervised machine learning. These techniques consider suspiciousness scores of program elements as *features* and train a model that ranks defective elements higher than non-defective elements. Such approaches require a training dataset of program elements annotated with suspiciousness scores computed using different FL techniques; each element needs to be manually labeled with the ground truth, as "defective" or "not-defective". Given a new defect, the trained model uses the FL scores computed for that defect to rank the elements. Training such models requires a large, annotated dataset, which can be hard to create. Further, training such models is typically computationally expensive [20], and performance depends heavily on the training dataset and features [181].

We propose an unsupervised approach that requires no training. We formulate the problem of combining multiple FL techniques as a rank aggregation (RA) [160] problem. RA involves combining multiple ranked lists (base rankers) into one ranked list (aggregated ranker) [63]. The RA problem has been studied extensively in

information retrieval [68], marketing and advertisement research [160], social choice (elections) [68], and genomics [130]. We propose to use RA algorithms to combine multiple FL techniques' ranked lists of suspicious statements.

We next describe our approach to combine FL techniques (Section 5.4.1) and using it to combine Blues and SBFL (Section 5.4.2).

5.4.1 RAFL: Rank Aggregation-Based Fault Localization

FL techniques typically assign suspiciousness scores to hundreds of program statements. Combining multiple ranked lists, which are often inconsistent, such that the result is as close as possible to the individual lists according to some distance metric, can become combinatorially intractable. For example, a brute force approach to create an ordered list of 100 statements by combining two lists of 500 statements, such that 800 of the statements are unique, will potentially require evaluating $\binom{800}{100} = 3.4 \times 10^{129}$ possible lists! Using a brute force method to compute this is computationally infeasible.

We propose rank aggregation-based FL (RAFL), a novel approach that uses RA algorithms to combine FL. Our technique is inspired from the research in search-based software engineering [98], which involves applying metahueristic search techniques to solve problems of balancing competing (and some times inconsistent) constraints. RAFL works as follows. Let $L_1, L_2, \ldots, L_m$ be m ordered lists of program statements (e.g., obtained using m FL techniques). RAFL aims to create an ordered list δ of length k that combines the statements in the individual lists by minimizing the weighted sum of the distances between δ and the individual lists. Formally, RAFL minimizes the objective function:

$$f(\delta) = \sum_{i=1}^{m} w_i d(\delta, L_i)$$

where w_i is the importance weight associated with list L_i, and d is a distance metric. To do this, RAFL samples multiple lists of k statements from the unique statements in the individual lists, using an algorithm-specific sampling strategy. For instance, the

cross-entropy Monte Carlo algorithm (CE) uses the multinomial sampling method that selects a statement to be part of a sampled list based on the statement's selection probability for that list. On the other hand, the genetic algorithm (GA) randomly creates *popSize* (specified as input) number of sampled lists and uses weighted random sampling to select the sampled lists where the weight for a list is determined based on its fitness (the objective function score). RAFL computes the objective function for each sampled list. Iteratively, RAFL updates the sampled list using the objective function computations based on the choice of RA algorithm used. For example, the CE algorithm updates the multinomial sampling probabilities of unique statements for sampled lists such that the updated sampled lists created based on the updated sampling probabilities of unique statements minimize their objective function score. On the other hand, the GA algorithm performs cross-over and mutation operations on the sampled lists such that the new generation of sampled lists produced as an outcome of these operations have lower objective function scores. This iteration continues until RAFL observes no change in the objective function scores for a fixed number of iterations, returning the lowest-scoring list.

Our RAFL implementation is build on the RankAggreg [221] package in R, which implements several RA algorithms (cross-entropy Monte Carlo algorithm (CE), genetic algorithm (GA), and a brute force algorithm) and provides distance metrics (Spearman Footrule [26], and Kendall's tau [25]). The brute force algorithm is advised to be used when combining lists of smaller size ($k \leq 10$), while CE and GA should be used for larger size lists. The left two columns in Figure 5.4 list RAFL configuration parameters, which can be used to select combinations of RA algorithms and distance metrics to combine FL. In our study, we use the default values of algorithm-specific parameters defined in the RankAggreg [221], which are tuned on independent dataset.

parameter	definition	SBIR value
k	size of the combined list	100
seed	seed specified for reproducibility	1
distance	Spearman or Kendall	Spearman
method	algorithm (CE or GA)	CE
maxIter	max # of iterations allowed (default 1000)	1000
convIn	# of consecutive iterations to decide if algorithm has converged (default: 7 for CE, 30 for GA)	7
N	#samples to be generated in each iteration (recommended to be at least k^2)	10,000
ρ	$(\rho \cdot N)$ is quantile of candidate lists sorted by the objective function scores used by the CE	0.01 if $N \geq 100$ and 0.1 otherwise
popSize	population size in each generation for the GA (default 100)	NA
CP	Cross-over probability for the GA (default 0.4)	NA
MP	Mutation probability for the GA	NA

Figure 5.4: RAFL configuration parameters.

5.4.2 SBIR: Combining SBFL and IRFL (Blues)

To combine the FL results from Blues (Section 5.2) and our SBFL (Section 5.3), we use the RAFL approach to develop SBIR using the cross-entropy Monte Carlo (CE) RA algorithm with the Spearman Footrule distance. We make these choices because prior work found CE to be typically more efficient than genetic algorithms [220] and than Borda count [58, 221], and because computing the Spearman Footrule distance is faster than Kendall's tau.

The CE algorithm represents an ordered list of k statements using a 0–1 matrix of size $n \times k$, where n is the total number of unique statements in the ranked lists and k is the length of the desired combined list. The algorithm imposes two constraints: each column sums up to exactly 1, and each row sums up to at most 1. Under this representation, an ordered list of size k is uniquely determined by reordering the matrix' rows (statements) such that the top k rows form the identity (that is, the first statement is the one with the 1 in the leftmost column, and so on). For example,

if the full list of unique statements was [S1, S2, S3, S4] ($n = 4$), then the following matrix:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

would translate to the top-3 candidate list of [S2, S4, S1].

The goal of the CE algorithm is to identify a matrix that results in the minimum objective function score out of all possible matrices. The CE algorithm uses the following four steps:

1. **Initialization** creates a probability matrix of size $n \times k$ and assigns a probability value of $\frac{1}{n}$ to all the cells of the matrix. This matrix represents the multinomial sampling probabilities of the statements: any one of the statements (rows) is equally likely to be in any one of the k positions. Next, CE runs steps 2 and 3 iteratively.

2. **Sampling** generates N 0–1 matrices using the restricted (truncated) multinomial sampling using the current probabilities [247] . The output of this step are N (new) randomly generated 0–1 matrices of size $n \times k$.

3. **Updating** computes the objective function scores for each of the N sampled matrices, sorts the sampled matrices in the ascending order of the scores, and identifies ρ-quantiles y^t of the sorted matrices. The algorithm uses the objective function scores of the matrices in iteration t to update the multinomial cell probabilities of unique elements that tend to minimize the objective function scores of the matrices sampled in the next iteration, as follows:

$$p_{jr}^{t+1} = (1-w)p_{jr}^{t} + w\frac{\sum_{i=1}^{N} I(f(\delta_i) \leq y^t)x_{ijr}}{\sum_{i=1}^{N} I(f(\delta_i) \leq y^t)}$$

where $1 \leq j \leq n$, $1 \leq r \leq k$, p_{jr}^{t} is the probability of the unique statement at the jr^{th} position in the matrix at iteration t and p_{jr}^{t+1} is its updated value at iteration $t+1$; $f(\delta_i)$ is the objective function score of the i^{th} sampled matrix and x_{ijr} is the value of the jr^{th} cell of the i^{th} sampled matrix; w is a weight parameter with a default value of 0.25 (from RankAggreg [221]) and I is the indicator function.

4. **Convergence** stops the iteration when the minimum value of the objective function does not change in a preset number of iterations. The matrix with a minimum objective function score in the final iteration represents the final combined list of statements.

Our SBIR implementation combines the ranked suspicious statement lists from SBFL and Blues, to produce a single list of top-100 statements. The right column in Figure 5.4 shows the values of configuration parameters we used to develop SBIR. We set importance weight $w_i = 1.0$ to assign equal importance to SBFL and Blues, and use default values of other parameters including w (used in updating sampling probabilities), and ρ that are tuned by prior work [220] on independent datasets. We select $k = 100$ because most APR tools consider at most 100 statements in their repair processes, and for efficiency: SBIR takes 10 min per defect, on average and combining FL results, SBIR took one week to execute for all the 815 defects in our evaluation (Section 5.6). In the final ranked list, SBIR assigns normalized suspiciousness scores varying from $[0.1, 1.0]$ to the statements where a statement at the top gets a score of 1.0 and subsequent statements get scores reduced by 0.1.

5.5 Subjects of Investigation and Evaluation Metrics

This section describes the real-world defects and the metrics we use to evaluate the performance of FL techniques.

5.5.1 Defect Benchmark

We use the latest version (version 2.0) of the Defects4J [87] benchmark that targets Java 8 and consists of 835 reproducible defects from 17 large open-source Java projects. Each defect comes with (1) one defective and one developer-repaired version of the project code with the changes minimized to those relevant to the defect; (2) a set of developer-written tests, all of which pass on the developer-repaired version and at least one of which evidences the defect by failing on the defective version; (3) the infrastructure to generate tests using modern automated test generation tools; and (4) defect information, including the bug report URL. Out of the 835 defects, 817 have the bug report URL available, making IRFL possible. For 815 of the 817 defects, the test execution information was relevant to make SBFL possible. We use these 815 defects to evaluate our FL techniques. As most APR tools are applicable to defects that involve modifying contiguous source code statements typically in a single source file, we characterize these 815 defects using the *Defect Complexity* parameters described in Section 2.2.3 of Chapter 2. For 689 of the 815 defects, developer-written patches modify a single source file (single-file-edit defects) while for 129 defects, developer-written patches modify a single source statement (single-line-edit defects) to repair the defect. Figure 5.5 shows the distribution of all 815 defects, single-file defects, and single-line defects across the 17 Defects4J projects.

5.5.2 Metrics to Evaluate Fault Localization Performance

We use the following two metrics, which are commonly used to evaluate the performance of FL techniques [320]:

identifier	project	description	all	sfd	sld
Chart	jfreechart	framework to create charts	8	8	4
Cli	commons-cli	API for parsing command line options	39	32	3
Closure	closure-compiler	JavaScript compiler	174	137	23
Codec	commons-codec	implementations of encoders & decoders	18	14	8
Collections	commons-collections	Java Collections Framework extensions	4	4	1
Compress	commons-compress	API for file compression utilities	47	43	4
Csv	commons-csv	API to read and write CSV files	16	15	5
Gson	gson	API to convert Java Objects into JSON	18	16	2
JacksonCore	jackson-core	core part of the Java JSON API (Jackson)	26	19	3
JacksonDatabind	jackson-databind	data-binding package for Jackson	111	91	13
JacksonXml	jackson-dataformat-xml	data format extension for Jackson	6	6	1
Jsoup	jsoup	HTML parser	93	75	18
JxPath	commons-jxpath	XPath (an expression language) interpreter	22	13	1
Lang	commons-lang	extensions to the Java Lang API	64	64	10
Math	commons-math	library of mathematical utilities	106	98	23
Mockito	mockito	a unit-test mocking framework	38	33	7
Time	joda-time	date and time processing library	25	21	3
total			815	689	129

Figure 5.5: The "all" column shows the 815 defects from the 17 real-world Java projects in the Defects4J (v2.0) benchmark localized using our FL techniques. The "sfd" column shows the 689 single-file-edit defects and the "sld" column shows the 129 single-line-edit defects.

1. *hit@k* counts the number of defects localized within the top-k ranked statements. It tells us how useful an FL technique is for an APR tool that considers the top k ranked statements. A higher value of *hit@k* provides APR tools an opportunity to repair more defects.

2. EXAM measures the fraction of ranked statements one has to inspect before finding a defective statement. A smaller value of EXAM score means the defective statements are ranked higher, providing APR tools an opportunity to produce more correct patches.

For consistency with prior studies [132, 164, 320], we consider a defect successfully localized when at least one of the developer-patch-modified statements is in the top-k statements. Unlike studies that break ties by reassigning average rank [215] or expected rank [320], we rank same-suspiciousness statements in the order they appear in the FL results, as this is how APR tools process them.

5.6 Evaluating Fault Localization and Key Findings

In this section, we first summarize the key findings of this study in terms of the research questions we ask. We then describe the evaluation details that lead to the summarized findings. We answer the following three research questions:

RQ1 Does Blues localize defects better than existing approaches?

Answer: Despite requiring no training data, for APR relevant scenarios ($k \geq 25$), Blues performs better than the state-of-the-art, *supervised*, statement-level IRFL technique used in iFixR. For example, Blues localizes 63% defects while iFixR localizes 61% defects in top-100 ranked statements. Further, Blues outperforms a statement-level BLUiR implementation that does not consider ranks of suspicious files: Blues localizes 37% defects while BLUiR localizes 30% defects in top-100 ranked statements. Considering the fraction of ranked suspicious statements required to be inspected to find defective statements averaged over localized defects, Blues outperforms both iFixR (Blues 3.4% vs iFixR 5.1%) and BLUiR (Blues 11.1% vs BLUiR 16.4%).

RQ2 Does SBIR, our combination of SBFL and Blues, localize defects better than SBFL and Blues?

Answer: For all ranked suspicious statement list sizes we investigated ($k \in \{1, 25, 50, 100\}$), SBIR consistently outperforms underlying SBFL and Blues by localizing more defects and placing defective statements higher in the ranked lists. For example,

considering top-100 suspicious statements, SBIR correctly identifies defective statements for 557 of the 815 (68.3%) defects, whereas SBFL does so for 546 (66.9%) and Blues for 304 (37.3%).

RQ3 Does SBIR outperform state-of-the-art FL?

Answer: SBIR consistently localizes more defects and places defective statements higher in the ranked lists than (1) 9 standalone FL techniques, (2) a supervised, learning-to-rank technique used by existing supervised combining FL techniques, and (3) a baseline combiner created using a simple method. Further, when SBIR is added to CombineFL [320], a state-of-the-art supervised technique that combines multiple FL techniques into one, CombineFL's performance consistently improves.

5.6.1 Blues' Evaluation

RQ1: Does Blues localize defects better than existing approaches?

Following sections compare the performance of Blues to localize defects in real-world programs with the state-of-the-art (Section 5.6.1.1) and baseline (Section 5.6.1.2) IRFL technique.

5.6.1.1 Blues's Comparison With the State-Of-The-Art

Figure 5.6 compares Blues with state-of-the-art statement-level IRFL technique used in iFixR [132] on the 171 Lang and Math project defects in Defects4J on which iFixR was evaluated[1]. As shown in Figure 5.6, considering ranked lists of size $\geq$ 25 (relevant for APR), Blues consistently localizes more defects (higher *hit@k*) than iFixR's IRFL . Comparing the fraction of ranked statements required to

[1]The iFixR FL results available at `https://github.com/TruX-DTF/iFixR/tree/master/data/stmtLoc` contain multiple statements with the same rank and multiple ranks for the same statement. We break ties by assigning the highest possible rank to each statement.

(171 defects)	*hit@k*					EXAM
	$k = 1$	25	50	100	all	k = all
iFixR	**26**	74	93	104	132	0.051
Blues	11	**79**	**97**	**107**	**149**	**0.034**

Figure 5.6: Comparing the FL performance of Blues with the state-of-the-art. For ranked suspicious statement lists of size ≥ 25, Blues localizes more defects (*hit@k*) and places buggy statements higher in the list (lower EXAM) than the state-of-the-art IRFL technique used in iFixR when evaluated on 171 Lang and Math defects in the Defects4J on which original iFixR was evaluated.

be inspected to identify defective statements for the localized defects, Blues places defective statements higher (lowering EXAM) in the ranked lists than iFixR. Blues' advantage of using a lightweight unsupervised approach outweighs iFixR's supervised technique that requires 6 file-level IRFL techniques.

5.6.1.2 Blues's Comparison With a Baseline

We implement a version of statement-level BLUiR (vanilla BLUiR) that does not consider the suspiciousness scores of the source files and instead ranks the suspicious statements only based on their similarity to the bug reports. Figure 5.7 compares Blues's and vanilla BLUiR performance on the 815 defects. As shown, Blues consistently outperforms vanilla BLUiR by localizing more defects (higher *hit@k*) and ranking defective statements higher (lower EXAM), for all list sizes we consider.

For APR relevant scenarios (ranked suspicious statement lists of size ≥ 25), Blues consistently localizes more defects and places defective statements higher in the ranked lists than the state-of-the-art, *supervised*, statement-level IRFL technique used in iFixR. Further, Blues outperforms a statement-level BLUiR implementation that does not consider ranks of suspicious files.

(815 defects)	*hit@k*					EXAM
	$k = 1$	25	50	100	all	$k =$ all
vanilla BLUiR	22	135	187	243	591	0.164
Blues	**26**	**183**	**238**	**304**	**593**	**0.111**

Figure 5.7: Comparing FL performance of Blues's with baseline. Blues significantly localizes more defects (higher *hit@k*) than vanilla BLUiR, which does not consider suspicious file scores for all k values and ranks defective statements higher (lower EXAM) in the ranked lists when compared on the 815 defects in the Defects4J v2.0.

5.6.2 SBIR's Evaluation

We evaluate SBIR on the 815 defects and two metrics from Section 5.5. This section compares the FL performance of SBIR with the underlying SBFL and Blues (Section 5.6.2.1) and the state-of-the-art FL techniques (Section 5.6.2.2). As SBIR computes ranked suspicious statement lists of size 100, for this analysis we consider ranked lists of size at most 100.

5.6.2.1 SBIR's Comparison With the Underlying Techniques

RQ2: Does SBIR, our combination of SBFL and Blues, localize defects better than SBFL and Blues?

Figure 5.8 shows the FL performance of SBIR, SBFL, and Blues for different list sizes we consider. As shown, for all list sizes, SBIR consistently localizes more defects (higher *hit@k*) and places defective statements higher (lower EXAM) in the ranked lists of localized defects than SBFL and Blues. For example, considering top-100 ranked suspicious statement lists, SBIR localizes 11 more defects than SBFL and 253 more defects than Blues. Comparing the ranks of defective statements in the top-100 ranked lists of defects localized, SBIR places defective statements at 20th position (EXAM 0.197) while SBFL places them at 23rd (EXAM 0.230) and Blues at 28th (EXAM 0.284) positions, on average. These results confirm prior findings suggesting

(815 defects)	*hit@k*				EXAM		
	$k = 1$	25	50	100	$k = 25$	50	100
SBFL	88	405	472	546	0.308	0.254	0.230
Blues	26	183	238	304	0.369	0.315	0.284
SBIR	**90**	**412**	**484**	**557**	**0.269**	**0.223**	**0.197**

Figure 5.8: Comparing FL performance of SBIR with underlying SBFL and Blues on 815 defects in Defects4J (v2.0). For all list sizes, SBIR consistently localizes more defects (higher *hit@k*) and places buggy statements higher in the list (lower EXAM) than underlying SBFL and Blues.

that combining FL techniques can lead to better FL [109, 140, 155, 255, 301, 320]. Thus, an APR tool using SBIR gets earlier opportunities to patch the defective statements and a more diverse set of localized defects than using SBFL or Blues.

For all list sizes we consider, SBIR consistently localizes more defects and places defective statements higher in the ranked lists than underlying SBFL and Blues.

5.6.2.2 SBIR's Comparison With the State-Of-The-Art

RQ3: Does SBIR outperform state-of-the-art FL?

We first evaluate SBIR with respect to standalone FL techniques. Next, we compare SBIR with a learning-to-rank supervised approach implemented using RankSVM [135], which is used by multiple existing combining FL techniques. We then compare SBIR against a baseline that we implement using Algorithm 1 to combine SBFL and Blues. Finally, we show that when added to supervised techniques that combine multiple standalone techniques into one, SBIR improves the results.

SBIR vs. standalone techniques. Our evaluation considers techniques that are evaluated on Defects4J, make no assumptions about a priori knowing the defective file, and localize defective statements (as opposed to methods or files). We compare SBIR with 9 standalone FL techniques, used in a recent FL evaluation [320]: two SBFL—

Ochiai and DStar; two mutation-based FL (MBFL) — Metallaxis and MUSE; three slicing — union, intersection, and frequency; one stack trace FL; and one predicate switching FL evaluated using CombineFL [320], a state-of-the-art FL framework. The existing evaluation [321] provides a dataset of the 357 defects of Defects4J (v1.0) annotated with suspiciousness scores of the 9 techniques but does not release the implementations of the individual techniques. We use the annotated suspiciousness scores to create ranked lists of statements for each of the 9 techniques. Of the 357 defects, 334 have bug reports available for which SBIR could be computed therefore, we consider these 334 defects for this analysis. Figure 5.9 compares the FL performance of the 9 techniques that includes our SBFL implementation (Ochiai using GZoltar v1.7.2), and SBIR in terms of the $hit@k$ and EXAM. As shown, SBIR consistently localizes more defects (higher $hit@k$) than al the 9 techniques. Considering the fraction of ranked suspicious statements required to be inspected to find defective ones averaged over localized defects, SBIR again outperforms (lower EXAM) all the 9 techniques.

SBIR vs, supervised combining FL techniques. Supervised combining FL techniques use learning-to-rank [32] machine learning approaches to train a model for a ranking task and combine a large array of standalone FL techniques. However, these techniques require a large, labeled dataset of program statements annotated with suspiciousness scores of multiple FL techniques and the ground truth of whether a statement is defective or not. Creating such a dataset of real-world defects can be hard and may restrict the use of trained models on arbitrarily selected projects (e.g., most state-of-the-art supervised combining FL techniques including CombineFL [320], DeepRL4FL [158], DeepFL [155], Fluccs [255], Savant [140], MULTRIC [301], and TraPT [157] use the Defects4J benchmark for training, so their trained models can not be used as-is to improve and evaluate APR on the Defects4J benchmark as the results would overfit) therefore, we develop RAFL, that does not require any training dataset.

(334 defects)		*hit@k*				EXAM
family	technique	$k = 1$	25	50	100	$k = 100$
SBFL	Ochiai	27	162	191	218	0.272
	DStar	29	162	193	219	0.273
MBFL	Metallaxis	40	148	168	185	0.232
	MUSE	23	91	99	111	0.188
slicing	slicing-union	19	84	95	106	0.471
	slicing-intersection	17	70	78	88	0.488
	slicing-frequency	19	83	95	107	0.462
stack trace	stack trace	13	23	23	23	0.660
predicate switching	predicate switching	9	23	23	23	0.650
	SBIR	**41**	**174**	**203**	**226**	**0.181**

Figure 5.9: Comparing FL performance of SBIR with 9 standalone FL techniques on 334 defects in Defects4J (v1.0). For all list sizes, SBIR consistently localizes more defects (higher *hit@k*) and places defective statements higher in the ranked lists (lower EXAM) than all of the 9 techniques.

As learning to rank is a de facto standard used by existing combining FL techniques, we compare SBIR with the results of combining SBFL and Blues using a learning to rank technique used by multiple combining FL techniques as described next.

We compare SBIR with RankSVM [135], a supervised approach that implements pairwise learning to rank model to combine ranked lists and that has been used by existing combining FL techniques such as CombineFL [320], Fluccs [255], and Savant [140]. For this, we first create an annotated dataset of the 815 defects by annotating program statements of each defect with normalized suspiciousness scores obtained using our SBFL and Blues along with the ground truth information. We then use this annotated dataset to train a model using the RankSVM that considers SBFL's and Blues's scores as features. To evaluate the trained model we use the combineFL framework [321] that uses 10–fold cross validation and computes $E_{inspect}@k$ and EXAM metrics. The $E_{inspect}@k$ metric break ties by computing the expected rank of defective

(815 defects)	$E_{inspect}$@k				EXAM$_{inspect}$
	$k = 1$	25	50	100	$k = 100$
SBIR (RankSVM)	69	365	433	505	0.209
SBIR (RAFL)	**69**	**383**	**454**	**509**	**0.188**

Figure 5.10: Comparing FL performance of SBIR with a learning-to-rank supervised technique used by multiple combining FL techniques on 815 defects from Defects4J (v2.0). For all lists of size $>$ 1, SBIR consistently localizes more defects (higher $E_{inspect}$@k) and places defective statements higher in the ranked lists (lower EXAM) than combining SBFL and Blues using supervised RankSVM.

statement in the ranked lists and then counts the number of defects whose defective statements have expected rank $\leq k$. Similarly, the EXAM scores are computed using expected ranks therefore, we denote it as EXAM$_{expect}$. Figure 5.10 compares the FL performance of SBIR with SBFL and Blues combined using RankSVM. As shown, for all lists of size > 1, SBIR consistently localizes more defects (higher hit@k) and places defective statements higher (lower EXAM$_{expect}$) in the ranked lists than RankSVM. The advantage of using the unsupervised RAFL that does not require any training dataset outweighs using supervised RankSVM.

SBIR vs. baseline. We implement a baseline (vanilla SBIR) by combining Blues with SBFL using the Algorithm 1. When combining ranked lists obtained using FL techniques using simple strategies that involve comparing suspiciousness scores such as Algorithm 1 then it must be noted that the suspiciousness scores computed by different FL techniques may not be directly comparable. Therefore, it is necessary to normalize the scores of all statements in the ranked lists before combining. Algorithm 1 computes the maximum normalized suspiciousness score assigned to each statement considering both the lists and sorts the statements based on the decreasing scores. The final ranked list is obtained by sorting the statements based on their maximum suspiciousness scores and breaking the ties by placing a statement that occurs in both lists higher than the one that occurs in one of the lists. Figure 5.11 compares

(815 defects)	*hit@k*				EXAM
	$k = 1$	25	50	100	k =100
vanilla SBIR	61	222	273	332	0.240
SBIR	**91**	**414**	**485**	**560**	**0.198**

Figure 5.11: Comparing FL performance of SBIR with a baseline on 815 defects in Defects4J (v2.0). For all list sizes, SBIR consistently localizes more defects (higher *hit@k*) and placing defective statements higher (lower EXAM) in the ranked lists than a baseline created by combining SBFL and Blues using Algorithm 1.

the FL performance of SBIR with this baseline on the 815 defects. As shown, SBIR consistently outperforms vanilla SBIR by localizing more defects (higher *hit@k*) and ranking defective statements higher (lower EXAM) in the ranked lists.

Adding SBIR to supervised combining FL techniques. We further evaluate whether adding SBIR to the set of standalone techniques existing supervised approaches combine improves overall performance. CombineFL [320] and DeepFL [155] outperform other approaches that include Fluccs [255], Savant [140], MULTRIC [301], and TraPT [157]. DeepFL's [156] implementation is public, but its data is not, making it impossible for us to integrate SBIR with the underlying techniques for a proper comparison. CombineFL makes public its data [321] for 9 techniques from 5 FL families (SBFL (Ochiai, DStar), MBFL (Metallaxis, MUSE), slicing-based (union, intersection, frequency), stack-trace-based FL, and predicate-switching-based FL; data for BugLocator and Bugspots is not released). We are, thus, able to evaluate adding SBIR to those 9 techniques, directly using CombineFL's implementation [321] that uses RankSVM [135] to combine these techniques. To perform this analysis, we use CombineFL's dataset of 357 defects from Defects4J (v1.0) that contains program statements covered by one or more of the 9 techniques annotated using the suspiciousness scores of those techniques. We augment this dataset by annotating each statement with the suspiciousness scores obtained using SBIR as well as add statements that are were missing from the original annotated dataset but were covered

		$E_{inspect}@k$				$\text{EXAM}_{inspect}$
technique	time	$k = 1$	25	50	100	$k = 100$
ST	seconds	11	23	23	23	0.665
ST + **SBIR**		**40**	**175**	**205**	**227**	**0.178**
ST + S + SBFL	minutes	29	175	202	228	0.250
ST + S + SBFL + **SBIR**		**46**	**187**	**213**	**231**	**0.158**
ST + S + SBFL + PS	∼10 min	28	178	208	230	0.243
ST + S + SBFL + PS + **SBIR**		**45**	**187**	**212**	**234**	**0.160**
ST + S + SBFL + PS + MBFL	hours	44	190	212	239	0.211
ST + S + SBFL + PS + MBFL + **SBIR**		**55**	**201**	**220**	**244**	**0.148**

Figure 5.12: Comparing FL performance of adding SBIR in a supervised technique using 334 defects in Defects4J (v1.0). "time" indicates the efficiency of standalone FL techniques considered for combining. Adding SBIR to the five families of FL techniques (ST = stack trace, S = slicing-based (union, intersection, frequency), SBFL = (Ochiai, DStar), MBFL = (Metallaxis, MUSE), and PS = predicate switching) improves the EXAM score in all cases, and increases the number of defects localized.

by SBIR. We annotate a statement with score 0.0 when that statement is not covered by an FL technique. As SBIR could compute ranked suspicious statement lists for 334 of the 357 defects for which both Blues and SBIR produce ranked lists, we use these 334 defects to perform this analysis. We follow the original evaluations [320] by training RankSVM model and using 10–fold cross validation to compute $E_{inspect}@k$ and $\text{EXAM}_{inspect}$ scores to analyze how adding SBIR to the set of techniques being combined affects FL performance. As shown in Figure 5.12, adding SBIR consistently localizes more defects (higher $E_{inspect}@k$) for all ranked suspicious statements lists sizes we consider. Further, in all combinations, adding SBIR enables CombineFL to rank defective statements higher (lower $\text{EXAM}_{inspect}$) in the ranked lists.

SBIR outperforms 9 standalone FL techniques, a supervised learning-to-rank technique used in multiple combining FL techniques, and a baseline created using a simple combining approach. Further, integrating SBIR consistently improves FL performance of a supervised technique that combines multiple standalone techniques.

5.7 Discussion

Recent studies have shown the effect of using different technologies, assumptions, and adaptations of test-suite-based FL techniques on the performance of repair tools [5, 11, 110, 164, 263, 286, 304]. Often, program repair researchers omit FL tuning used by their repair tools while presenting repair performance, which leads to bias in comparing performance of different repair tools [164]. Further, the FL implementations are often tightly coupled to the repair tool implementations, which makes it hard to use the FL for other repair tools, or improve the FL. Our FL techniques can be used to mitigate this bias as they can serve as a plugin by future repair tools to decouple their FL implementations from their repair algorithm implementation, as is done in some frameworks, including JaRFly [200].

The main goal of our study is not to develop a combining FL technique that outperforms existing supervised ones. Our goal is to combine FL to test the hypothesis that using combined SBFL and IRFL improves APR quality. We could, in theory, use existing supervised techniques, but we didn't because most existing techniques (e.g., DeepFL [155], Fluccs [255], Savant [140], MULTRIC [301], TraPT [157]) do not use bug-report-based IRFL and use Defects4J benchmark for training, so using their trained models cannot be used as-is for APR evaluation on Defects4J. To re-train the models used by these supervised techniques we require a large, labeled bug dataset independent of projects used in the Defects4J. Instead of this, we create a novel unsupervised technique requiring no training data. While we show that RAFL outperforms RankSVM, which is used in multiple existing combining FL techniques, analyzing if RAFL outperforms all other learning-to-rank approaches is out of scope for this study.

5.8 Threats to Validity

We address the threat to construct and content validity by using standard FL evaluation metrics that measure both, if a defect is localized or not within the top-k ranked statements ($hit@k$) and how precise is the ranking of the statements (EXAM). Further, unlike some of the existing studies that break ties between ranked statements by reassigning average rank [215] or expected rank [320], we rank same-suspiciousness statements in the order they appear in the FL results, as this is a more realistic usage. We address the threat to internal validity by reusing the publicly available implementations of rank aggregation algorithms instead of reimplementing them. We address the threat to external validity by using Defects4J (v2.0) that has significantly more projects and defects than earlier versions. We make all code and data available to help others reproduce our results.

5.9 Contributions

In this study, we created Blues, the first statement-level, IRFL technique that outperforms the state-of-the-art without needing training dataset, and is reusable for APR. We created RAFL, a novel unsupervised technique to combine multiple FL techniques and SBIR, an FL technique that uses both bug reports and tests to localize defects, which consistently localizes more defects and places defective statements higher in the ranked suspicious statement lists than underlying FL techniques and the state-of-the-art. Finally, we have released the open-source implementations of Blues and RAFL, and evaluation of SBIR, Blues, and SBFL on Defects4J (v2.0), showing that using SBIR outperforms the other FL techniques and state-of-the-art FL.

This work is joint with Yuriy Brun, and credit for this work is shared between the two of us. All of our data, source code, and documentation to produce the results presented in this study are available at `https://bit.ly/3fs3d99` and the pre-print version of this study [198] is available at `https://arxiv.org/abs/2011.08340`.

CHAPTER 6

BETTER AUTOMATIC PROGRAM REPAIR BY USING BUG REPORTS AND TESTS TOGETHER

6.1 Introduction

A recent study [166] shows that using perfect FL (manually specifying the defective program locations) enables repair tools to patch more defects. We also made the observation that manually improving FL accuracy improves APR quality for SOSRepair and enables it to correctly patch significantly more defects (from 41% to 70%) (recall Section 3.3.4 in Chapter 3). However, manually improving FL for APR is unrealistic in practice and therefore in this chapter, we investigate whether using a more precise automated FL technique improves APR quality.

Analyzing the cause of why APR tools produce incorrect or plausible patches (that overfit the tests used during the repair process) or not produce any patch at all even though a correct patch exists in the search space revealed that APR tools sometimes: (1) produce plausible patches because of using non-defective program statements ranked higher than defective statements in the output ranked list of FL technique used by repair tools or (2) do not produce any patch at all either because they timeout while attempting to construct patches using non-defective statements that are ranked higher in the list, or the list does not provide *all* the defective statements are required to construct a correct patch. These are known as localization errors in the APR literature and are investigated in recent studies [132, 164] as the problem of FL bias in APR. We hypothesize that APR tools using an FL technique, which identifies and places defective statements higher in the ranked list of suspicious program statements, should improve APR quality. The study described in this chapter tests this hypothesis.

While there exists a variety of FL techniques that use different kinds of defect information sources (recall Chapter 5), most APR tools typically use spectrum-based FL (SBFL) techniques that uses test execution coverage to compute the suspiciousness scores of program elements such as classes, methods, and statements. The elements are ranked based on their scores, and repair tools use top-ranked (typically ≤ 100) elements to patch defects. Most repair tools use SBFL because they rely on test suites, which are readily available for defects. While FL researchers are actively working on improving FL by using multiple defect information sources, to the best of our knowledge, there does not exist any repair tool that uses these advancements to localize defects in their repair process. The work closest to experimenting with FL in APR is a repair technique iFixR [132] that internally uses bug reports and an information-retrieval-based FL (IRFL) technique to localize defects. iFixR patches defects that 16 SBFL-based repair tools cannot, and vice versa.

Because combining FL techniques that use different information sources (e.g., test execution coverage, bug reports, and stack trace) can significantly outperform underlying individual FL techniques [140, 155, 320], we, therefore, test our hypothesis by using combined SBFL and IRFL (SBIR) in APR. To the best of our knowledge, this is the first investigation of the effect of combined FL on APR. To test our hypothesis, we use our FL techniques (SBFL, Blues, and SBIR) described in Chapter 5. Instead of creating a new repair techniques, we experiment with using multiple existing state-of-the-art APR tools, which enables us to evaluate the effectiveness of our proposed approach across different types of repair tools.

This chapter is organized as follows. Section 6.2 describes the APR tools we evaluate. Section 6.3 describes the real-world defects, metrics used to evaluate repair quality, and our methodology to execute repair experiments using multiple FL techniques. Section 6.4 describes the key findings and results in terms of the research questions we ask. Section 6.5 describes two case studies of real-world defects showing how using

the combined FL improves repair quality. Section 6.6 discusses the assumptions we make in this study. Section 6.7 addresses the threats to validity of our results and Section 6.8 summarizes our contributions.

6.2 Program Repair Using Multiple Fault Localization

Our APR tool selection criteria require that tools apply to general defects, rather than specialized, and have public implementations available so that they can be customized to take precomputed FL results. Instead of developing a new APR tool or arbitrarily selecting tools from state-of-the-art, we select Arja [313] and SimFix [111] that are the most ($Sen = 66.9\%$) and least ($Sen = 29.5\%$) FL-sensitive general purpose repair tools out of the 11 APR tools evaluated in a recent study [166] for their FL sensitiveness. We select a third tool, SequenceR [41], which uses fundamentally different repair approach than Arja and SimFix, and whose FL-sensitiveness ($Sen = 39.5\%$) lies between Arja and SimFix. Although there are more effective APR tools (e.g., CURE [113]) than SequenceR, which is only applicable to single-line-edit defects, we use SequenceR because its implementation is public and can be customized. We next describe details of the three APR tools and how we customize them.

Arja [313] uses genetic-programming-based [133] repair approach to construct patches, and is the most FL-sensitive ($Sen = 66.9\%$) general purpose APR tool of the 11 APR tools evaluated for FL sensitivity [166]. It effectively prunes the search space of candidate patches by using a novel patch representation for genetic programming, multi-objective search, test filtering procedure, type matching, and several other strategies. Arja implementation [312] uses GZoltar (v0.1.1) test execution framework to implement SBFL with Ochiai technique. It takes as input: (1) path to the directory of source code, (2) path to the directory of compiled classes of the source code, (3) path to the directory of compiled classes of test code, and (4) paths to the dependencies (jar files), and optionally, configuration parameters such path to the

directory containing GZoltar output. We customize Arja's internal FL implementation to read precomputed FL result from a file instead of executing and processing the output of GZoltar. We build a wrapper over Arja's implementation to execute it using multiple FL techniques and the Defects4J benchmark. Our wrapper takes as input project name and bug id (associated with each defect in Defects4J), and the path to the FL results file computed using different FL techniques, and executes Arja using these inputs. The original Arja evaluation [313] considered statement with suspiciousness score higher than 0.1 to repair a defect. We set this threshold to 0 to make Arja attempt to contruct patches using all 100 suspicious program statements.

SimFix [111] uses fix pattern mining-based [165] repair approach to construct patches. It is the least ($Sen = 29.5\%$) FL sensitive general purpose APR tool of the 11 repair tools evaluated for FL sensitivity [166]. The SimFix implementation [108] is hard-coded to work with the Defects4J benchmark. Its input is a project name and bug id (associated with each defect in Defects4J), and, optionally, precomputed statement-level FL results. (Without providing FL results, SimFix runs SBFL with Ochiai implemented using GZoltar (v1.6), employing a test purification technique [302] to improve FL accuracy). To patch a defect, SimFix uses code patterns mined from frequently occurring code changes in developer-written patches. SimFix identifies code snippets similar to the suspicious code, defining similarity using structural properties, variable names, and method names. SimFix ranks the identified code snippets based on the number of mined patterns applied to replace the buggy code, and then selects the snippets (one at a time) from the ranked list of the top 100, applies the pattern-based modifications to produce a candidate patch, and validates the patch against the purified failing tests. SimFix can stop once a patch passes the test suite [111] but its implementation [108] generates all the patches that pass at least one of the purified failing tests. In this study, we use only the patches that pass *all* of the developer tests provided with the defect. Similar to customizing Arja's implementation, we customize

SimFix to make it read from the precomputed FL result files. We build a wrapper over SimFix implementation to execute it using multiple FL techniques and the Defects4J benchmark. Our wrapper takes as input Defects4J project name, bug id, and path to the file storing precomputed FL result, and executes SimFix using these inputs. The original SimFix evaluation [111] does not specify the number of suspicious statements it used, but its implementation [108] uses top-200 suspicious program statements. To fairly evaluate SimFix concerning all three FL techniques (SBFL, Blues, and SBIR), we limit it to using top-100 statements of SBFL and Blues.

SequenceR [41] uses neural machine translation-based [272] repair approach to construct patches. Its FL sensitiveness ($Sen = 39.5\%$) lies in between Arja and SimFix. To patch a defect, it uses a pre-trained model trained on 35,578 single-line commits, carefully curated from commits in open-source projects. The model takes as input a defective line location in a defective source file, abstracts the defect context around the defective line, and *translates* the defective line into a fixed line. SequenceR implementation [40] takes as inputs: path to the pre-trained model, path to a defective source file, line number indicating where the defect is in the source file, beam size for prediction (default value = 50), and output directory to store the generated patches. The original SequenceR evaluation [41] used (manually created) perfect FL and estimates its efficiency using top-10 suspicious statements to repair a defect. SequenceR's implementation requires defective statement location as input. Therefore, in our study, we execute SequenceR with perfect FL and record the information about defects it patches plausibly or correctly. For each correctly patched defect, we check if the list of top-10 ranked suspicious statements obtained using multiple FL techniques includes the defective statement modified by SequenceR to produce a plausible or correct patch. If an FL technique does not localize the modified defective statement in its top-10 ranked list, SequenceR can not patch that defect using that FL technique.

6.3 Subjects of Investigation and Experimental Methodology

This section describes the real-world defects used in our experiments, the metric we use to evaluate repair quality, and the methodology of executing repair experiments using different APR tools and FL techniques.

6.3.1 Defect Benchmark

We use the defects available in the latest version of Defects4J (v2.0) benchmark for this study (recall Section 5.5). Because manually assessing the correctness of automatically produced patches that modify multiple files is error-prone and suffers from bias [141, 308], we consider the 689 single-file-edit defects from the 815 defects described in Section 5.5. Our subset includes defects from all the Defects4J projects. As SequenceR is applicable on single-line-edit defects, we use the 129 single-line-edit defects that are a subset of the 689 defects.

6.3.2 Metric to Evaluate Repair Quality

Prior repair tools' evaluations that measure patch correctness use either manual inspection [141, 184, 297] or automatically-generated evaluation test suites [5, 141, 200, 296, 298] (recall Chapter 3). While manual inspection is subjective and could be biased, using low-quality evaluation test-suites could inaccurately measure patch correctness [141]. In Chapter 3, we described a methodology to generate *high-quality* evaluation test suites to measure patch correctness. In this study, we use that methodology and refine it by incorporating manual inspection of produced patches as described next. Our novel patch evaluation methodology enables us to reap the benefits of both the methods of evaluating the patch correctness.

For each patch, we consider the developer-written patch (available for all Defects4J defects) as an oracle, and use EvoSuite [81] to generate 10 test-suites using 10 seeds, a search budget of 12 minutes per seed, and a coverage criterion of maximizing line

coverage of the developer-modified classes. This methodology is the state-of-the-art objective (but potentially incomplete [141]) automated test-driven patch correctness methodology [200]. To evaluate the correctness of a patch, we first execute the held-out evaluation tests on the patch. If a test fails, we annotate such patch as *plausible* (the term used for a patch that passes developer-written tests but is incorrect [227]). Otherwise, we manually inspect the patch and compare it against the developer's patch. If the patch is semantically equivalent to the developer's patch, we annotate it as *correct*. If it is not, we annotate it as *plausible*. If a patch is partially correct or we cannot determine its semantic equivalence to developer patch because it requires extensive domain knowledge, which often happens when the modifications are made in different methods, we conservatively annotate the patch as *plausible*, but keep a record of such scenarios. Thus, our patch evaluation methodology is conservative as we only consider a patch to be *correct* if it passes all evaluation tests and is semantically equivalent to the developer's patch. To study the effect of improving FL on APR, we compare the defects a repair tool *correctly* patches out of all the defects attempted, using different FL techniques.

6.3.3 Experimental Methodology

Using the dataset described in Section 6.3.1, we use Arja and SimFix to repair 689 single-file-edit defects and SequenceR to repair 129 single-line-edit defects using SBFL, Blues, and SBIR for FL. We execute each repair tool separately using SBFL, Blues, and SBIR. The patches produced by repair tools using different FL techniques are first automatically evaluated to filter out patches that modify files different from developer modified ones. Next, the remaining patches are evaluated for correctness using the methodology described in Section 6.3.2. We do not otherwise modify the implementations of APR tools described in Section 6.2 except customizing them to use our precomputed FL results.

We run our experiments using a cluster of 50 compute nodes, each with a Xeon E5-2680 v4 CPU with 28 cores (2 processors, 14 cores each) running at 2.40GHz. Each node had 128GB of RAM and 200GB of local SSD. We launched multiple repair attempts in parallel, each requesting 2 cores on one compute node. We specify a timeout of 24 hours per FL techniques for each repair attempt to ensure that repair tools can try out all suspicious statements. Note that the focus of this study is repair quality (how many defects can be correctly repaired) and not repair efficiency (how quickly defects can be repaired).

6.4 Evaluating Repair Quality and Key Findings

In this section, we first summarize the key findings of this study in terms of the research questions we ask. We then describe the evaluation details that lead to the summarized findings.

We answer the following three research questions.

RQ1 Does SBIR, our combination of SBFL and Blues, improve APR quality?

Answer: Yes. SBIR improves the quality of Arja and SequenceR, more FL-sensitive tools, and enables correctly repairing some defects that they do not fix using other FL. For less FL-sensitive SimFix, using SBIR leads to the same repair quality as using SBFL but more than using Blues.

RQ2 Does APR using Blues outperform state-of-the-art IRFL-based APR?

Answer: The tools are complementary. Arja correctly patches 9 defects, 8 of which iFixR fails to patch. SequenceR correctly patches 2 defects, 1 of which iFixR fails to patch. SimFix correctly patches 7 defects, 6 of which iFixR fails to patch. iFixR correctly repairs 8 defects, 5 of which none of the other three tools repair.

RQ3 Does overuse of the Defects4J benchmark affect our study?

Answer: Recent research has found that results from old Defects4J versions do not generalize to new defects [66]. We confirm this finding. For example, Arja correctly repairs 3–5% of the 343 old Defects4J defects, but only 1–3% of the 346 new defects. However, SBIR improves APR quality on old and new defects, supporting the generalizability of our results.

We next describe the results in detail for each of the above listed research questions.

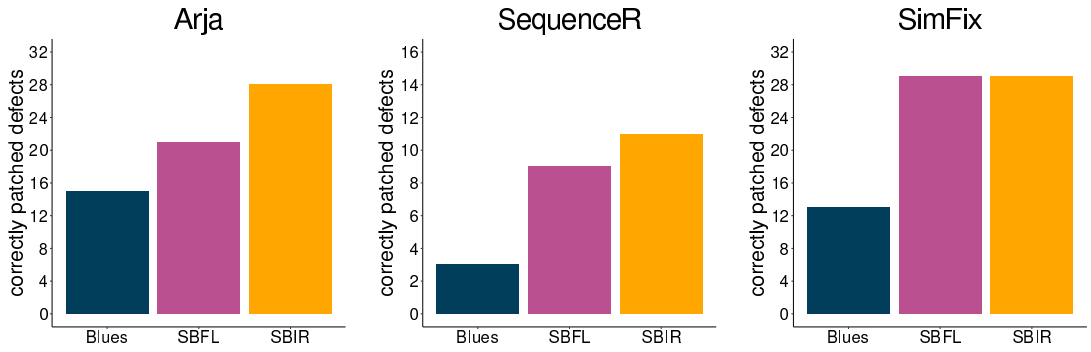
6.4.1 Effect of Using SBIR on Repair Quality

RQ1: Does SBIR, our combination of SBFL and Blues, improve APR tools?

The top of Figure 6.1 compares repair quality of the three repair tools using the three FL techniques. As shown, Arja and SequenceR correctly patch more defects when using SBIR than when using SBFL or Blues. Specifically, Arja using SBIR correctly repairs 7 (33.3%) more defects than using SBFL and 13 (86.7%) more defects than using Blues. SequenceR using SBIR correctly patches 2 (22.2%) (out of a smaller subset of single-line defects) more defects than using SBFL and 8 (266.7%) more defects than using Blues. SimFix unsurprisingly correctly patches the same number of defects when using SBFL but 17 (130.8%) more defects than using Blues.

More FL-sensitive repair tools, Arja and SequenceR, correctly patch complementary defects using SBFL and Blues, as evident by the row showing the union of defects they patch using SBFL and Blues. However, as the less FL-sensitive SimFix uses test case purification [302] and expands each suspicious statement at most by ± 5 lines to address inaccurate FL, it does not patch complementary defects.

As described earlier, one of the the reason why repair tools produce incorrect patches or do not produce any patch at all is that the tools attempt to construct patches using irrelevant suspicious statements placed before the defective ones in the ranked FL results. This phenomenon is known as the FL bias or localization error in



	Arja (689 defects)	SequenceR (129 defects)	SimFix (689 defects)
repair quality assessment			
SBFL	21	10	30
Blues	15	4	13
SBFL ∪ Blues	25	12	30
SBIR	**28**	**12**	**30**
localization error assessment			
perfect FL	21	24	21
upper bound	35	24	32
↓ # of defects not correctly patched due to localization error ↓			
SBFL	14	14	2
Blues	20	20	19
SBIR	**7**	**12**	**2**

Figure 6.1: The effect of using combined FL on repair quality. SBIR improves repair quality and reduces localization errors for more FL-sensitive APR tools. Arja and SequenceR, more FL-sensitive tools, correctly patch complementary defects using SBFL and Blues, and benefit more from using SBIR. SimFix, a less FL-sensitive repair tool, correctly patches the same defects using SBIR as SBFL but more than Blues.

the APR literature. We compare the usefulness of SBFL, Blues, and SBIR in reducing the localization error in APR. For this, we execute each of the three repair tools using perfect FL (by providing only the defective statement locations) and measure their quality. As shown in Figure 6.1, for Arja and SimFix, which can use multiple suspicious statements to patch multi-line bugs, using a perfect FL patches fewer defects than when using SBIR. We found that this happens because their repair algorithms fail to construct correct patches for some defects in the absence of the non-defective statements that are closely related to the defective ones. We next compute the total

number of defects a repair tool can correctly patch by taking the union of defects correctly patched using the perfect FL and using the three FL techniques. We call this optimistic number of correctly patched defects an "upper bound", the term used in a prior study [166]. The localization error of an FL technique is then the difference between the upper bound and the number of defects correctly patched using that FL technique. As shown in Figure 6.1, localization error is significantly reduced using SBIR, specially for more FL-sensitive repair tools, than using SBFL or Blues.

Overall, Arja and SequenceR, significantly benefit from SBIR. Arja using SBIR correctly patches 28 (80%) of the 36 upper bound defects, whereas using SBFL, it only patches 21 (60%) and using Blues only 15 (43%). SequenceR using SBIR correctly patches 12 (50%) of the 24 upper bound defects, whereas using SBFL, it patches 10 (42%) and using Blues 4 (17%). SimFix, correctly patches 30 (94%) of the 32 upper bound defects using both SBIR and SBFL, and patches 13 (41%) using Blues.

Repair quality using SBIR vs. union of SBFL and Blues. This tells us whether using SBIR provides any benefit over executing repair tools separately using SBFL and Blues and combining their results. Arja patches a total of 25 defects (row 3 in Figure 6.1) using SBFL and Blues. These 25 defects include 4 defects (Compress 27, Jsoup 33, Jsoup 55, and Time 15) that Arja could not patch using SBIR. However, Arja using SBIR patches 28 defects that include 7 defects (Closure 78, Gson 7, Jsoup 39, Jsoup 68, Jsoup 85, JxPath 5, and Lang 7) that Arja could not patch using SBFL or Blues. Thus, for Arja, SBIR is even more beneficial than using both SBFL and Blues. SequenceR patches 12 defects using both SBFL and Blues. These 12 defects include one defect (Cli 40) that SequenceR could not patch using SBIR. However, SequenceR using SBIR also patches 12 defects that includes one defect (JacksonCore 25) that SequenceR could not patch using SBFL or Blues. Thus, SBIR provides the same benefit to SequenceR as using both SBFL and Blues. SimFix correctly patches 30 defects using both SBFL and Blues. These 30 defects are the

same as defects correctly patched using SBIR. Thus, for SimFix, using SBIR provides the same benefit as using just SBFL or both SBFL and Blues.

Repair quality using SBIR vs. original published versions. Arja using SBIR correctly patches 4 defects (Lang-7, Lang-10, Lang-59, and Math-35) original Arja did not. Of the defects in our dataset, the original Arja correctly patched 15 defects [313] (plus 3 others that either had no bug reports (Chart-3) or were multi-file-edit defects (Math-22 and Math-98)). Of these 15, Arja using SBIR correctly patches 12, but not the other 3 (Lang-35, Math-39, and Math-86). We examined the original evaluation's patches[1] and found that for these 3 defects, Arja had produced only a single patch, which is highly uncommon for Arja (it produced many patches for all other defects it patched), suggesting that there is something special about these defects or the process the Arja evaluation followed in repairing them. Overall, Arja with SBIR correctly patches 1 more defect than the original Arja. SimFix using SBIR correctly patches 3 defects (Closure-68, Closure 92, and Closure-126) original SimFix did not. Of the defects in our dataset, original SimFix correctly patched 21 defects [111] (plus 6 others that either had no bug reports (Chart-3, Chart-7, Chart-20) or were multi-file-edit defects (Closure-63, Math-71, and Math-98)). Of these 22, SimFix with SBIR correctly patches 19. (Note that the original evaluation [111] listed 7 more defects (Closure-115, Lang-16, Lang-27, Lang-39, Lang-41, Lang-50, and Lang-60) as patched correctly. The authors subsequently identified one of those (Lang-27) as incorrect[2], and our analysis revealed that the six others are also incorrect. SimFix with SBIR could not patch the remaining two defects (Math-35 and Math-63). Overall, SimFix with SBIR correctly patches 1 more defect than the original SimFix. SequenceR's original evaluation used perfect FL [41], so a direct comparison is not appropriate. With perfect FL, original SequenceR patched 14 defects correctly, and with SBIR,

[1] `https://github.com/yyxhdy/defects4j-patches/tree/master/Arja`

[2] `https://github.com/xgdsmileboy/SimFix/tree/master/final/result`

it patches 6 (Chart-11, Closure-73, Closure-86, Lang-59, Math-58, and Math-75) of those 14.

> SBIR significantly improves repair quality and reduces localization errors for more FL-sensitive APR tools and enables correctly repairing some defects that they cannot repair with other FL techniques. For less FL-sensitive APR, it provides the same repair quality as other FL techniques.

6.4.2 Comparing Repair Quality Using Blues With the State-Of-The-Art

RQ2: Does APR using Blues outperform state-of-the-art IRFL-based APR?

We compare the defects correctly patched by Arja, SequenceR, and SimFix using Blues against the published evaluation of iFixR [132], the state-of-the-art IRFL-based APR tool, on the 156 defects from the Lang and Math projects (the only ones used in iFixR's evaluation [132]). iFixR correctly patches 8 of the 156 defects. Arja correctly patches 9, SimFix 7, and SequenceR only 2 (out of a smaller subset of single-line defects). Figure 6.2 shows the defects correctly patched by the APR tools. The techniques produce complementary patches. For example, only 1 (Math-35) out of 9 defects correctly patched by Arja is patched by iFixR. Similarly, only 1 (Math-75) out of 2 defects correctly patched by SequenceR is patched by iFixR, and only 1 (Math-57) out of 7 defects correctly patched using SimFix is patched by iFixR. And only three (Math-35, Math-57, and Math-75) of 8 defects correctly patched by iFixR are patched using our APR tools with Blues.

> We conclude that iFixR (the state-of-the-art IRFL-based repair tool) and repair tools using Blues correctly patch a comparable number of complementary defects.

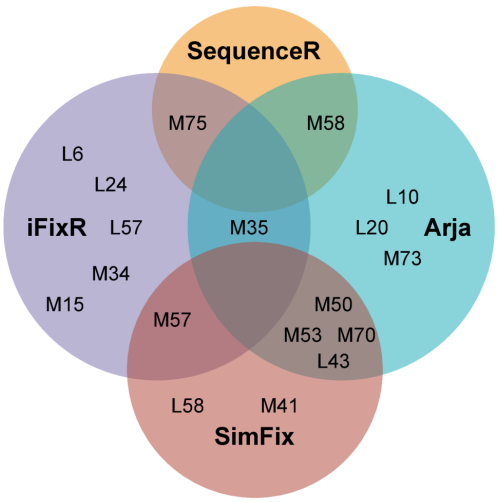


Figure 6.2: Comparison of repair quality using Blues with the state-of-the-art. Comparing the correctly patched defects from the 156 Lang (L) and Math (M) project defects by Arja, SequenceR, and SimFix using Blues against the ones patched by iFixR [132] shows complementary results.

6.4.3 Comparing Repair Quality Across New and Old Defects4J

RQ3: Does overuse of the Defects4J benchmark affect our study?

Figure 6.3 compares the repair quality of the three APR tools using three FL techniques across 17 projects available in Defects4J (v2.0) along with the old and new defects added in Defects4J. The bottom of Figure 6.3 shows evidence that past APR evaluations of Arja and SimFix overfit to the defects on which they were evaluated, and fail to generalize to new defects. SequenceR, which can patch only single-line-edit defects, generalizes better than Arja and SimFix. Comparing repair quality on defects in the older version of the Defects4J with the newly added defects in Defects4J (v2.0) shows that the repair quality is significantly higher on old defects than on new ones. This is consistent with a prior study [66] that shows that existing evaluations of repair tools overfit on older Defects4J versions. For example, Arja correctly patches 3–5% (3% Blues, 4% when using SBFL, 5% SBIR) of the 343 defects in the older version of Defects4J, but only 1–3% (1% Blues, 2% SBFL, 3% SBIR) of the 346 *new* defects. Similarly, SimFix correctly patches 3–6% (3% Blues, 6%

project	Arja (689 defects)			SequenceR (129 defects)			SimFix (689 defects)		
	Blues	**SBFL**	**SBIR**	**Blues**	**SBFL**	**SBIR**	**Blues**	**SBFL**	**SBIR**
Chart	1	2	2	0	1	1	1	1	1
Closure	1	2	3	0	1	1	2	7	7
Lang	3	4	6	1	1	1	2	3	3
Math	6	6	7	2	2	3	5	10	10
Mockito	0	0	0	0	0	0	0	0	0
Time	0	1	0	0	0	0	0	1	1
↓ new defects added in Defects4J (v2.0) ↓									
Cli	1	1	1	0	3	2	0	0	0
Closure	0	0	0	0	0	0	0	1	1
Codec	2	1	2	0	2	2	0	0	0
Collections	0	0	0	0	0	0	0	0	0
Compress	0	1	0	1	0	1	0	1	1
Csv	0	0	0	0	0	0	1	1	1
Gson	0	0	1	0	0	0	1	1	1
JacksonCore	0	0	0	0	0	1	0	0	0
JacksonDatabind	0	0	0	0	0	0	0	2	2
JacksonXml	0	0	0	0	0	0	0	0	0
Jsoup	1	3	5	0	0	0	0	1	1
JxPath	0	0	1	0	0	0	1	1	1
all defects	$\frac{15}{689} = 2.2\%$	$\frac{21}{689} = 3.1\%$	$\mathbf{\frac{28}{689} = 4.1\%}$	$\frac{4}{129} = 3.1\%$	$\frac{10}{129} = 7.8\%$	$\mathbf{\frac{12}{129} = 9.3\%}$	$\frac{13}{689} = 1.9\%$	$\frac{30}{689} = 4.3\%$	$\mathbf{\frac{30}{689} = 4.3\%}$
old defects	$\frac{11}{343} = 3.2\%$	$\frac{15}{343} = 4.4\%$	$\mathbf{\frac{18}{343} = 5.2\%}$	$\frac{3}{69} = 4.3\%$	$\frac{5}{69} = 7.2\%$	$\mathbf{\frac{6}{69} = 8.7\%}$	$\frac{10}{343} = 2.9\%$	$\frac{22}{343} = 6.4\%$	$\mathbf{\frac{22}{343} = 6.4\%}$
new defects	$\frac{4}{346} = 1.2\%$	$\frac{6}{346} = 1.7\%$	$\mathbf{\frac{10}{346} = 2.9\%}$	$\frac{1}{60} = 1.7\%$	$\frac{5}{60} = 8.3\%$	$\mathbf{\frac{6}{60} = 10.0\%}$	$\frac{3}{346} = 0.9\%$	$\frac{8}{346} = 2.3\%$	$\mathbf{\frac{8}{346} = 2.3\%}$

Figure 6.3: Comparing repair quality across old and new defects in Defects4J. The figure compares the repair quality of Arja, SequenceR, and SimFix using SBFL, Blues, and SBIR. SequenceR is evaluated on the 129 single-line-edit defects while Arja and SimFix are evaluated on 689 single-file-edit defects. Arja and SimFix correctly patch significantly more defects in older version of Defects4J than new defects added in Defects4J (v2.0). SimFix, while can patch only single-line-edit defects, generalizes better than Arja and SimFix. Using SBIR improves the quality of all repair tools on both old defects and new defects showing that our results generalize.

when using SBFL, 6% SBIR) of the 343 defects in the older version of Defects4J, but only 1–2% (1% Blues, 2% SBFL, 2% SBIR) of the 346 *new* defects. SequenceR gives comparable performance by correctly patching 4–9% (4% when using Blues,7% SBFL, 9% SBIR) of the 69 defects in the older version of Defects4J and 2–10% (2% Blues, 8% SBFL, 10% SBIR) of the 60 *new* defects. Nevertheless, using SBIR improves the repair quality on both new and old defects showing that our results generalize.

We conclude that although repair quality overfits to the older Defects4J versions, using SBIR improves the repair quality on both old and new defects added in Defects4J (v2.0).

6.5 Case Studies

This section describes two real-world examples from our experiments to illustrate how SBIR helps in reducing localization errors and improving quality of APR.

6.5.1 Example-1: Codec-5 Defect

This example illustrates how a repair tool using SBFL produces a plausible patch using non-defective program statements before the defective statements, whereas when using Blues and SBIR, repair tool generates a correct patch using actual defective statement.

The Codec-5 defect in Defects4J v2.0 involves repairing a null pointer exception that occurs while using the `decode` method in which the code accesses `buffer` that is `null` or has reached end of file.[3] Figure 6.4 shows the defective `decode` method along with actual line numbers and the statements localized by the three FL techniques along with the ranks of the localized statements in their respective ranked lists of suspicious statements. Figure 6.5(a) shows the developer patch for the defect that adds a null-pointer check. To correctly repair this defect, the null-pointer check must be inserted *before* the `switch` statement on line#588 because both cases of the switch-statement modify `buffer`. Thus, developer adds it on line#586. However, the check could also be added *before* line#551, 554, 557, 585, 587, or 588 resulting in semantically equivalent programs.

RAFL takes as input the top 100 suspicious statements obtained by applying the SBFL and Blues to localize the defect. The two lists have 32 statements in common providing SBIR with 168 unique statements from which it needs to select and rank the top 100 statements that are as close as possible (measured using Spearman distance metric) to both SBFL's and Blues' lists. Note that $\binom{168}{100} = 1.1 \times 10^{48}$ are a lot of

[3] `https://issues.apache.org/jira/browse/CODEC-98`

```
550  void decode(byte[] in, int inPos, int inAvail){
551   if (eof) {                                                    //++ SBFL(70)
552      return;
553   }
554   if (inAvail < 0) {                                            //++ SBFL(71)
555      eof = true;
556   }
557   for (int i = 0; i < inAvail; i++) {                           //++ SBFL(72) Blues(17) SBIR(7)
558     if (buffer == null || buffer.length - pos < decodeSize) {
559       resizeBuffer();
560     }
561     byte b = in[inPos++];
562     if (b == PAD) {
563       // We're done.
564       eof = true;
565       break;
566     } else {
567       if (b >= 0 && b < DECODE_TABLE.length) {
568         int result = DECODE_TABLE[b];
569         if (result >= 0) {
570           modulus = (++modulus) % 4;
571           x = (x << 6) + result;
572           if (modulus == 0) {
573             buffer[pos++] = (byte) ((x >> 16) & MASK_8BITS);
574             buffer[pos++] = (byte) ((x >> 8) & MASK_8BITS);
575             buffer[pos++] = (byte) (x & MASK_8BITS);
576           }
577         }
578       }
579     }
580   }
581
582   // Two forms of EOF as far as base64 decoder is concerned: actual
583   // EOF (-1) and first time '=' character is encountered in stream.
584   // This approach makes the '=' padding characters completely optional.
585   if (eof && modulus != 0) {                                    //++ SBFL(73) Blues(45) SBIR(46)
586
587      x = x << 6;                                                //++ SBFL(56)
588      switch (modulus) {                                         //++ SBFL(57)
589         case 2 :
590            x = x << 6;
591            buffer[pos++] = (byte) ((x >> 16) & MASK_8BITS);
592            break;
593         case 3 :
594            buffer[pos++] = (byte) ((x >> 16) & MASK_8BITS);     // SBFL(42)
595            buffer[pos++] = (byte) ((x >> 8) & MASK_8BITS);
596            break;
597      }
598    }
599  }
```

Figure 6.4: Localization of program statements that can be used to repair the Codec-5 defect in Defects4J (v2.0) using the three FL techniques. The annotations "SBFL", "Blues", and "SBIR" in comments show which FL technique covers the statement in their respective ranked lists of the suspicious statements and the numbers inside parenthesis show the ranks of the statements in the respective lists. The method throws a null pointer exception on line#594 while updating `buffer`, which causes a developer test to fail. SBFL ranks this statement higher than all statements (annotated with"//++") that can be used to repair defect. Developer patch (Figure 6.5(a)) adds a null-pointer check on line#586.

```
@@ -583,9 +583,6
        // EOF (-1) and first time '=' character is encountered in stream.
        // This approach makes the '=' padding characters completely optional.
        if (eof && modulus != 0) {
+           if (buffer == null || buffer.length - pos < decodeSize) {
+               resizeBuffer();
+           }

            x = x << 6;
            switch (modulus) {
```

(a) Developer's patch for Codec-5 defect.

```
@@ -554,7 +554,10 @@
        if (inAvail < 0) {
            eof = true;
        }
-       for (int i = 0; i < inAvail; i++) {
+       if (buffer == null || buffer.length - pos < encodeSize) {
+                   resizeBuffer();
+               }
+               for (int i = 0; i < inAvail; i++) {
            if (buffer == null || buffer.length - pos < decodeSize) {
                resizeBuffer();
            }
```

(b) Correct patch produced by Arja using Blues for Codec-5 defect.

```
@@ -591,7 +591,18 @@
                    buffer[pos++] = (byte) ((x >> 16) & MASK_8BITS);
                    break;
                case 3 :
-                   buffer[pos++] = (byte) ((x >> 16) & MASK_8BITS);
+       if (buffer == null || buffer.length - pos < encodeSize) {
+                       resizeBuffer();
+               }
+                   buffer[pos++] = (byte) ((x >> 16) & MASK_8BITS);
+                   buffer[pos++] = (byte) ((x >> 8) & MASK_8BITS);
                    break;
            }
```

(c) Plausible patch produced by Arja using SBFL for Codec-5 defect.

```
@@ -579,7 +579,10 @@
        }
    }

-       // Two forms of EOF as far as base64 decoder is concerned: actual
+       if (buffer == null || buffer.length - pos < encodeSize) {
+                   resizeBuffer();
+               }
+               // Two forms of EOF as far as base64 decoder is concerned: actual
    // EOF (-1) and first time '=' character is encountered in stream.
    // This approach makes the '=' padding characters completely optional.
    if (eof && modulus != 0) {
```

(d) Correct patch produced by Arja using SBIR for Codec-5 defect.

Figure 6.5: Example illustrating how SBIR enables APR to correctly patch a defect that it patches plausibly when using SBFL. The figure shows patches for the Codec-5 defect in Defects4J (v2.0). (a) Developer's patch adds an null-pointer check *before* the switch-statement. (b) Arja using Blues correctly patches the defect by adding the check at a correct location (*before* the switch-statement). (c) Arja using SBFL *incorrectly* patches the defect by adding the check at an incorrect location (specific switch case that throws null pointer exception). (d) Arja using SBIR patches the defect correctly by adding the check at a correct location (*before* the switch-statement).

possible combinations to test and using a brute force method to find the solution is computationally infeasible. RAFL uses the CE algorithm to iteratively sample multiple lists containing 100 of the 168 unique suspicious statements that minimize the objective function defined in terms of the Spearman Footrule distance between the sampled lists and the input lists. RAFL takes 157 iterations to converge reducing the value of the objective function from 5346 to 4154. The top left plot in Figure 6.6 shows the decreasing objective function scores with each iteration and the top-right chart shows the histogram of the objective function scores in the last iteration. Based on these two plots, we can get a general idea about the rate of convergence and the distribution of candidate lists at the last iteration. The bottom chart in Figure 6.6 shows how the suspicious statements obtained in the final ranked list of SBIR are ranked in the SBFL and Blues's lists along with their average ranks. Comparing the

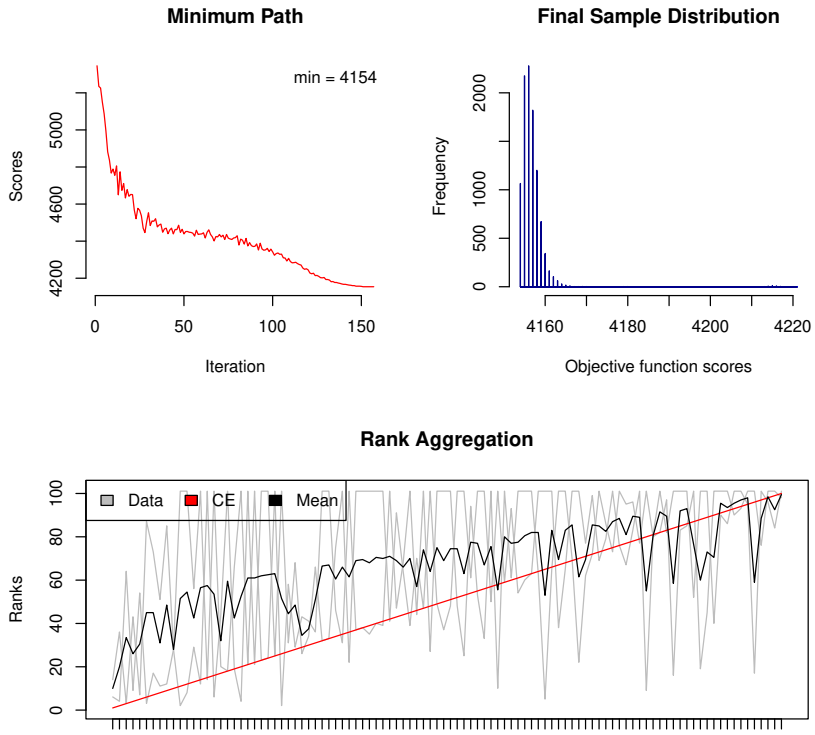


Figure 6.6: Using RAFL to combine SBFL's and Blues's FL results of Codec-5 defect in Defects4J (v2.0). RAFL combines the ranked lists of SBFL and Blues to produce top-100 ranked suspicious statement list. The top left chart shows how the objective function scores are minimized by RAFL and converge in 157 iterations placing a defective statement at the 7th position in SBIR's list. The top right chart shows the histogram of the objective function scores in the last iteration. The bottom chart shows how the statements in final SBIR's list are ranked in the SBFL and Blues's lists along with their average ranks.

ranks of defective statements (line#551, 554, 557, 585, 587, 588) where the correct patch can be applied to repair the defect in Figure 6.4, it can be realized that SBIR improves the ranks of defective statements, and places a defective statement (line#557) at the 7th position. Thus, if a repair tool considered only top-10 suspicious statements to repair the Codec-5 defect, it would likely find the correct patch only using SBIR but not using SBFL or Blues. Blues identifies a defective statement (line#557 in Figure 6.4) as the 17th most suspicious statement. Figure 6.5(b) shows the patch produced by Arja using Blues. As shown, the patch adds the same if-statement as the developer patch (Figure 6.5(a)) before line#557 instead of line#587. SBFL ranks a

non-defective statement (line#594 in Figure 6.4) at the 42nd position and places all the defective statements below this statement. This makes Arja using SBFL produce a plausible patch shown in Figure 6.5(c), which applies the correct modification to the non-defective statement. Even though the patch passes all the developer tests used in the repair process, it is *incorrect* because it only fixes the `case 3` of the `switch` statement that causes the null pointer exception. SBIR ranks a defective statement (line#557 in Figure 6.5(a)) as the 7th most suspicious statement. Figure 6.5(d) shows the patch produced by Arja using SBIR. The patch is correct and is semantically equivalent to the developer patch.

Note that all three patches constructed by Arja using SBFL, Blues, and SBIR use the correct modifications required to repair the defect. However, the location of the program where the patch is applied makes them *correct* or *plausible*. Thus, APR tools using the FL techniques that place the defective statements higher in the ranked list of suspicious statements are more likely to produce correct patches. This example shows that using SBIR, which ranks the defective statements higher than SBFL and Blues can enable repair tools to produce more correct patches.

6.5.2 Example-2: Closure-78 Defect

This example illustrates why a repair tool is unable to produce any patch for a defect when using SBFL or Blues whereas it correctly patches that defect using SBIR. The Closure-78 defect in the Defects4J v2.0 involves fixing the closure JavaScript compiler to return a null value when a number is divided by zero instead of throwing the JSC_DIVIDE_BY_0_ERROR error[4] because JavaScript allows divide by zero operations. Figure 6.7 shows a portion of the defective method `private Node performArithmeticOp(int opType, Node left, Node right)` that throws a divide

[4] `https://storage.googleapis.com/google-code-archive/v2/code.google.com/closure-compiler/issues/issue-381.json`

```
709  case Token.MOD:
710      if (rval == 0) {
711      error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right); //++ Blues(39)
             SBIR(39)
712      return null;
713      }
714      result = lval % rval;
715      break;
716  case Token.DIV:
717      if (rval == 0) {
718      error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right); //++ SBFL(1)
             Blues(37) SBIR(1)
719      return null;
720      }
721      result = lval / rval;
722      break;
```

Figure 6.7: Localization of program statements that can be used to repair the Closure-78 defect in Defects4J (v2.0) using the three FL techniques. The two non-consecutive defective statements annotated with "//++" cause the divide-by-zero error of Closure-78 defect. The annotations show which of the three FL techniques cover the statements and their ranks in the respective suspicious statement lists.

```
@@ -708,14 +708,12 @@
       break;
     case Token.MOD:
       if (rval == 0) {
-        error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right);
         return null;
       }
       result = lval % rval;
       break;
     case Token.DIV:
       if (rval == 0) {
-        error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right);
         return null;
       }
       result = lval / rval;
```

(a) Developer's patch for Closure-78 defect.

```
@@ -708,14 +708,12 @@
       break;
     case Token.MOD:
       if (rval == 0) {
-        error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right);
         return null;
       }
       result = lval % rval;
       break;
     case Token.DIV:
       if (rval == 0) {
-        error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right);
         return null;
       }
       result = lval / rval;
```

(b) Correct patch produced by Arja using SBIR.

Figure 6.8: Example illustrating how SBIR enables APR to correctly patch a defect that it cannot patch when using SBFL and Blues. The left code snippet shows the developer patch that deletes defective program statements and the right shows the exact same patch produced by Arja only using SBIR.

by zero error while computing division and modulus operations. None of the existing 14 APR tools [164] can correctly or plausibly patch this defect. Most APR tools do not apply to this defect because the repair of this defect involves editing non-consecutive statements. As Arja uses multi-objective genetic programming-based repair approach, it can potentially repair this defect. Figure 6.8(a) shows the developer-written patch for the defect, which deletes the two defective statements.

Applying Blues identifies the defective statements, line#711 and line#718 in Figure 6.7, at the 39th and 37th positions respectively. Applying the SBFL technique

only identifies one of the defective statement (line#718 in Figure 6.7) at the 1st position. SBIR ranks the defective statements at the 1st (line#718) and 39th (line#711) positions. Arja using SBFL produces only a partially correct patch that deletes the line#718 while Arja using Blues did not produce any patch because it timed out while attempting to modify the 38 non-defective statements ranked higher in the Blues's list. Arja using SBIR produces a correct patch that is exactly the same as the developer patch shown in Figure 6.8(b). Thus, even though the FL evaluations consider identifying at least one defective statement as FL success, APR tools may not be able to repair a defect unless *all* defective statements are identified and are ranked higher in the list of suspicious statements.

6.6 Discussion

We assume that both bug-exposing tests and bug reports are available to repair tools. Our assumption may not always hold (e.g., several defects in Defects4J (v2.0) have no documented bug reports, and prior studies [121, 132] show that for 92% of defects, the bug-exposing tests are added after the bug is reported). However, existing repair tools cannot function without either a failing test or bug reports. Further, repair tools that use only bug reports are not fully automatic, as they require a human to validate their proposed patches. By contrast, test-driven APR can be fully automated but a large fraction of their patches are incorrect [200, 206, 252], meanwhile developers are likely not interested in APR with a human in the loop [207]. Our extending of repair tools to use SBIR, which uses both bug reports and test suites, enables repair tools to be fully automated and to produce better quality patches. This is a worthwhile achievement even if not all defects in industrial settings have the requisite artifacts, and may motivate developers to create the artifacts in the future, which reinforces an already recommended practice.

6.7 Threats to Validity

Arja, one of the APR tools we use in our study is stochastic and results may vary on runs. We mitigate this threat by running Arja two times for each of the three FL techniques. Our study's large computational requirements (running three repair tools using three FL techniques with a 24-hour timeout required eight weeks of wall-clock time on a cluster) making running many replications difficult. We make all code and data available to help others reproduce our results.

We address the threat to internal validity by reusing the publicly available implementations of repair tools instead of reimplementing them. We address the threat to external validity by selecting diverse repair tools and using Defects4J (v2.0), which has significantly more projects and defectsthan earlier versions.

6.8 Contributions

In this study, we present an evaluation of three state-of-the-art APR tools (Arja, SequenceR, and SimFix) that use fundamentally different repair algorithms using multiple FL techniques (SBIR, Blues, and SBFL) on Defects4J (v2.0) benchmark, showing that using the combined FL technique (SBIR) improves the repair performance of APR tools. Our results demonstrate that combining bug reports and tests leads to better FL, and enables higher-quality APR, even repairing defects that have not been repaired using existing FL. Our findings suggest further research to improve APR by combining bug-report-based, test-based, and other sources of bug information.

This work is joint with Yuriy Brun, and credit for this work is shared between the two of us. All of our data, source code, and documentation to produce the results presented in this study are available at `https://bit.ly/3fs3d99` and the pre-print version of this study [198] is available at `https://arxiv.org/abs/2011.08340`.

CHAPTER 7

RELATED WORK

This chapter describes the existing research work organized in the context of automatic program repair (Section 7.1), studies of repair quality and other properties of automatic program repair (Section 7.2), automated fault localization (Section 7.3), and automated test generation (Section 7.4).

7.1 Automatic Program Repair

APR techniques can be categorized into two types based on the kind of specifications they use to repair programs. The first category is the verification-based APR, which uses formally specified program specifications to repair programs. For example, these repair techniques use first-order logic specifications [61, 62], separation logic specifications [273], Hoare-style specifications [48, 222, 228], linear-temporal logic specifcations [76, 115], SAT-based specifications [92], deductive synthesis [127, 144], contracts [173], and model checking for Boolean programs [93, 245] to fix defects. The second category is the test-suite-based APR, which uses developer-written test suites as program specifications and produces a patch that passes all the tests in the test suite. As test suites are readily available and developers write tests more often than formal program specifications, APR predominantly uses developer-written test suites as specifications to fix defects. This dissertation focuses on such test-suite-based APR techniques.

The APR techniques typically follow a three-step process: identifying the location of a defect, producing candidate patches, and validating those patches. The method

used for each of these steps can significantly affect the tool's success. To improve APR, researchers have proposed to use different kinds of fault localization strategies [11, 110, 132, 178, 263, 304], patch generation algorithms (e.g., heuristics-based [111, 150, 177, 218, 269, 287], semantic-based [5, 95, 188, 275], and learning-based [41, 96, 244]), and patch validation methodologies [268, 277, 305, 308, 311].

We next describe the three classes of repair approaches to repairing defects using failing tests to identify faulty behavior and passing tests to encode desirable behavior: heuristics-based, semantic-based, and learning-based repair. The heuristics-based techniques use search-based software engineering [97] to generate many candidate patches and then validate them against tests. GenProg [147, 150, 284] uses a genetic programming heuristic [133] to search the space of candidate repairs. TrpAutoRepair [224] limits its patches to a single edit, uses random search instead of genetic programming, and heuristics to select which tests to run first, improving efficiency. Prophet [177] and HDRepair [145] automatically learn bug-fixing patterns from prior developer-written patches and use them to produce candidate patches for new defects. AE [282] is a deterministic technique that uses heuristic computation of program equivalence to prune the space of possible repairs, selectively choosing which tests to use to validate intermediate patch candidates. ErrDoc [269] uses insights obtained from a comprehensive study of error handling bugs in real-world C programs to automatically detect, diagnose, and repair the potential error handling bugs in C programs. JAID [39] uses automatically derived state abstractions from regular Java code without requiring any special annotations and employs them, similar to the contract-based techniques to generate candidate repairs for Java programs. Qlose [53] optimizes a program distance, a function of syntactic and semantic differences between the original defective and the patched programs, while generating candidate patches. DeepFix [96] and ELIXIR [244] use learned models to predict erroneous program locations along with patches. ssFix [296] uses existing code that is syntactically

related to the context of a bug to produce patches. CapGen [287] works at the AST node level and uses context and dependency similarity (instead of semantic similarity) between the suspicious code fragment and the candidate code snippets to produce patches. SapFix [183] and Getafix [249], two tools deployed on production code at Facebook, efficiently produce correct repairs for large real-world programs. SapFix [183] uses prioritized repair strategies, including pre-defined fix templates, mutation operators, and bug-triggering change reverting, to produce repairs in realtime. Getafix [249] learns fix patterns from past code changes to suggest repairs for bugs that are found by Infer, Facebook's in-house static analysis tool. SimFix [111] considers the variable name and method name similarity, as well as structural similarity between the suspicious code and candidate code snippets. Similar to CapGen, it prioritizes the candidate modifications by removing the ones that are found less frequently in existing patches. SketchFix [104] optimizes the candidate patch generation and evaluation by translating faulty programs to sketches (partial programs with holes) and lazily initializing the candidates of the sketches while validating them against the test execution. Par [123] and SOFix [168] use predefined repair templates to generate candidate patches. These repair templates are created based on the repair patterns mined from StackOverflow posts by comparing code samples in questions and answers for fine-grained modifications. Synthesis techniques can also construct new features from examples [45, 94], rather than address existing bugs.

The semantic-based techniques use semantic reasoning to synthesize patches to satisfy an inferred specification. Nopol [299], Semfix [204], DirectFix [189], and Angelix [190] use SMT or SAT constraints to encode test-based specifications. S3 [143] extends the semantics-based family to incorporate a set of ranking criteria such as the variation of the execution traces similar to Qlose [53]. JFIX [142] extends Angelix [190] to target Java programs. SemGraft [188] infers specifications by symbolically analyzing a correct reference implementation instead of using test

cases. Genesis [174], Refazer [239], NoFAQ [54], Sarfgen [275], and Clara [95] process correct patches to automatically infer code transformations to generate patches. SearchRepair [122] blurs the line between heuristics-based and semantic-based techniques by using constraint-based encoding of the desired behavior to replace suspicious code with semantically-similar human-written code from elsewhere.

The learning-based APR techniques (e.g., CURE [113], DeepFix [96], DeepRepair [289], SequenceR [40], CoCoNut [179]) frame APR as neural machine translation (NMT) problem (translating defective program into patched program similar to translating one language into another) and use modern deep-learning-based algorithms to produce candidate patches. For example, CoCoNut uses a context-aware NMT architecture that represents the defective program and its surrounding context separately, to automatically fix bugs in multiple programming languages. These techniques require additional training data (i.e, the tuples of defective program statements, context, and fixed program statements) to capture complex relations between defective and patched program.

The methods presented in this dissertation can be used to design a new APR technique as well as improve the existing techniques. We also present evaluation frameworks that aim to help researchers to properly evaluate their techniques' ability to produce high-quality patches for real-world defects. Our work enables properly comparing techniques with respect to patch quality, and encourages the creation of new techniques whose focus is producing high-quality patches on real-world defects. Empirical studies of fixes of real bugs in open-source projects can also improve repair techniques by helping designers select change operators and search strategies [119,318]. Understanding how repair techniques handles particular classes of errors, such as security vulnerabilities [150,217] can guide tool design. For this reason, some automated repair techniques focus on a particular defect class, such as buffer overruns [250, 251], unsafe integer use in C programs [46], single-variable atomicity violations [114],

deadlock and livelock defects [161], concurrency errors [167], and data input errors [7] while other techniques tackle generic bugs. Although our evaluation focused on tools that fix generic bugs, our methodology can be applied to focused repair as well.

7.2 Empirical Studies Evaluating Automatic Program Repair

Prior work has argued the importance of evaluating the types of defects APR techniques can repair [199], and evaluating the generated patches for understandability, correctness, and completeness [195]. Yet many of the prior evaluations of repair techniques have focused on what fraction of a set of defects the technique can produce patches for (e.g., [35, 52, 67, 114, 150, 184, 282, 284]), how quickly they produce patches (e.g., [147,282]), how maintainable the patches are (e.g., [83]), and how likely developers are to accept them (e.g., [1, 123]).

However, some recent studies have focused on evaluating the quality of repair and developing approaches to mitigate patch overfitting. For example, on 204 Eiffel defects, manual patch inspection showed that AutoFix produced high-quality patches for 51 (25%) of the defects, which corresponded to 59% of the patches it produced [216]. While AutoFix uses contracts to specify desired behavior, by contrast, the patch quality produced by techniques that use tests has been found to be much lower. Manual inspection of the patches produced by GenProg, TrpAutoRepair (referred to as RSRepair in that paper), and AE on a 105-defect subset of ManyBugs [227], and by GenProg, Nopol, and Kali on a 224-defect subset of Defects4J showed that patch quality is often lacking in automatically produced patches [184]. An automated evaluation approach that uses a second, independent test suite not used to produce the patch to evaluate the quality of the patch similarly showed that GenProg, TrpAutoRepair, and AE all produce patches that overfit to the supplied specification and fail to generalize to the intended specification [29, 252]. This work has led to new techniques that improve the quality of the patches [122, 175, 177, 296, 297, 311]. For example, DiffTGen

generates tests that exercise behavior differences between the defective version and a candidate patch, and uses a human oracle to rule out incorrect patches. This approach can filter out 49.4% of the overfitting patches [296]. Using heuristics to approximate oracles can generate more tests to filter out 56.3% of the overfitting patches [297]. UnsatGuided uses held-out tests to filter out overfitting patches for synthesis-based repair, and is effective for patches that introduce regressions but not for patches that only partially fix defects [311]. Automated test generation techniques that generate test inputs along with oracles [22, 90, 197, 265] or use behavioral domain constraints [9, 85], data constraints [75, 201, 202], or temporal constraints [15, 16, 18, 69, 208] as oracles could potentially address the limitations of the above-described approaches.

Using independent test suites to measure patch quality is imperfect, as test suites are partial and may identify some incorrect patches as correct. On a dataset of 189 patches produced by 8 repair techniques applied to 13 real-world Java projects, independent tests identify fewer than one fifth of the incorrect patches, underestimating the overfitting problem [141]. However, on other benchmarks, the results are much more positive. For example, on the QuixBugs benchmark, combining test-based and manual-inspection-based quality evaluation could identify 33 overfitting patches, while test-based evaluation alone identified 29 of the 33 (87.9%) [307]. While the human judgment is a criterion not used by the repair tools for patch construction, it is fundamentally different from the correctness criterion we use in our evaluation, as it is often difficult for humans to spot bugs even when told exactly where to look for them [214]. Further, using independently generated test suites instead of using the subset of the original test suite to evaluate patch quality ensures that we do not ignore regressions a patch is most likely to introduce. Poor-quality test suites result in patches that overfit to those suites [227]. Our evaluation goes further, demonstrating that high-quality, high-coverage test suites still lead to overfitting, and identifying other relationships between test suite properties and patch quality.

Our work has focused on understanding the effectiveness of repair techniques to patch large real-world Java programs correctly and to identify what factors affect the generation of high-quality patches. Studying the effects of test suite size, coverage, number of failing tests, and test provenance on the quality of the patches generated by Angelix on the IntroClass [149] and Codeflaws [266] benchmarks of defects in small programs finds results consistent with ours. By contrast, our work focuses on real-world defects in real-world projects and heuristics-based repair. Further, prior work has shown that the selection of test subjects (defects) can introduce evaluation bias [21,223]. Our evaluation focuses precisely on the limits and potential of repair techniques on a large dataset of defects, and controls for a variety of potential confounds, addressing some of Monperrus' concerns [195].

7.3 Automated Fault Localization

Fault Localization [2, 27, 51, 117, 159, 196, 215, 240, 241] aims to precisely identify potential defective program elements that cause the defects to facilitate bug fixing. The most widely studied class of fault localization techniques is spectrum-based fault localization (SBFL) which usually apply statistical analysis (e.g., Tarantula [117], Ochiai [2], and Ample [51]) or learning techniques [27, 240, 241, 242] to the execution traces of both passed and failed tests to identify the most suspicious program elements (e.g., statements/methods). The insight behind these techniques is that program elements primarily executed by failed tests are more suspicious than the elements primarily executed by passed tests. However, a program element executed by a failed test does not necessarily indicate that the element has impact on the test execution and has caused the test failure. To bridge the gap between coverage and impact information, researchers proposed mutation-based fault localization (MBFL) [196, 213, 315], which injects changes to each program element (based on mutation testing [60, 107]) to check its impact on the test outcomes. MBFL has been applied to both general bugs (e.g.,

Metallaxis [213]) and regression bugs (e.g., FIFL [315]). Besides SBFL and MBFL, researchers have proposed to utilize various other information for fault localization such as program slicing (e.g., [6, 234]) that use dynamic program dependencies, stack trace analysis (e.g., [291, 294]) to use error messages, predicate switching (e.g., [317]) to use test results from mutating the results of conditional expressions, information retrieval-based fault localization (IR–based FL) (e.g., [319], [243]) that use bug report information, and history-based fault localization (e.g., [124, 229]) that use the development history to identify the suspicious program elements that are likely to be defective.

Existing FL techniques can be classified into two categories. The first category is the standalone techniques. For example, PRoFL [178] improves SBFL using patch execution results from APR, PREDFL [109] uses runtime statistics from statistical debugging to improve SBFL, PRFL [316] uses the PageRank algorithm, XGB-FL [303] uses a classifier to learn the importance of program statements and features, such as execution sequence and semantics, and UniVal [136] uses execution profiles and the success and failure information from program executions, in conjunction with statistical inference. SBIR, our presented FL technique, outperforms the techniques that fall in this category. The second category is supervised techniques that combine multiple FL techniques, such as CombinedFL [320], DeepRL4FL [158], DeepFL [155], Fluccs [255], Savant [140], MULTRIC [301], and TraPT [157]. These techniques typically use *learning to rank* [32] algorithms such as RankSVM [135] that consider the suspiciousness scores from multiple FL techniques as features, and train a model to predict if a given program element is defective. SBIR outperforms RankSVM.

Fault Localization in Program Repair. Most program repair tools use SBFL implemented using off-the-shelf coverage tracking tools and the Ochiai ranking strategy [5,41,95,96,111,150,177,188,244,269,275,287]. R2Fix [162] and iFixR [132] are the only two IRFL-based repair tools, and no prior repair tool uses combined SBFL and

IRFL. Although, using patch-execution results from repair tools to refine FL results can outperform state-of-the-art SBFL and MBFL techniques [178]. Recent studies have shown the effect of using different technologies, assumptions, and adaptations of test-suite-based FL techniques on the performance of repair tools [5,11,110,164,263,286,304]. Often, program repair researchers omit FL tuning used by their repair tools while presenting repair performance, which leads to bias in comparing performance of different repair tools [164]. Further, the FL implementations are often tightly coupled to the repair tool implementations, which makes it hard to use the FL for other repair tools, or improve the FL. Our FL techniques can be used to mitigate this bias as they can serve as a plugin by future repair tools to decouple their FL implementations from their repair algorithm implementation, as is done in our JaRFly [200] framework.

7.4 Automated Test Generation

Test generation techniques that extract oracles from Javadoc specifications are the closest prior work to Swami, our proposed approach of automatically generating tests with oracles from natural language specifications. Toradacu [90] and Jdoctor [22] do this for exceptional behavior, and @tComment [265] for null pointers. These tools interface with EvoSuite or Randoop to reduce their false alarms. JDoctor, the latest such technique, combines pattern, lexical, and semantic matching to translate Javadoc comments into executable procedure specifications for pre-conditions, and normal and exceptional post-conditions. Our approach builds on these ideas but applies to more general specifications than Javadoc, with more complex natural language. Unlike these tools, Swami does not require access to the source code and generates tests only from the specification, while also handling boundary conditions. When structured specifications, Javadoc, and source code are all available, these techniques are likely complementary. Meanwhile, instead of generating tests, runtime verification of Java API specifications can discover bugs, but with high false-alarm rates [154].

Requirements tracing maps specifications, bug reports, and other artifacts to code elements [65, 101, 293], which is related to Swami's *Section Identification* using the Okapi model [237, 262]. Static analysis techniques typically rely on similar information-retrieval-based approaches as Swami, e.g., BLUiR [243], for identifying code relevant to a bug report. Swami's model is simpler, but works well in practice; recent studies have found it to outperform more complex models on both text and source code artifacts [230, 267].

Dynamic analysis can also aid tracing, e.g., in the way CERBERUS uses execution tracing and dependency pruning analysis [71]. Machine learning can aid tracing, e.g., via word embeddings to identify similarities between API documents, tutorials, and reference documents [309]. Unlike Swami, these approaches require large ground-truth training datasets. Future research will evaluate the impact of using more involved information retrieval models.

Automated test generation (e.g., EvoSuite [81] and Randoop [211]) and test fuzzing (e.g., afl [4]) generate test inputs. They require manually-specified oracles or oracles manually encoded in the code (e.g., assertions), or generate regression tests [81]. Swami's oracles can complement these techniques. Differential testing can also produce oracles by comparing behavior of multiple implementations of the same specification [28, 38, 77, 246, 257, 306] (e.g., comparing the behavior of Node.js to that of Rhino), but requires multiple implementations, whereas Swami requires none.

Specification mining techniques use execution data to infer (typically) FSM-based specifications [16, 17, 18, 89, 134, 138, 139, 169, 170, 171, 172, 208, 233, 248]. TAUTOKO uses such specifications to generate tests, e.g., of sequences of method invokations on a data structure [50], then iteratively improving the inferred model [50, 274]. These dynamic approaches rely on manually-written or simple oracles (e.g., the program should not crash) and are complementary to Swami, which uses natural language specifications to infer oracles. Work on generating tests for non-functional

properties, such as software fairness, relies on oracles inferred by observing system behavior, e.g., by asserting that the behavior on inputs differing in a controlled way should be sufficiently similar [85], [30], [9]. Meanwhile, assertions on system data can also act as oracles [201, 202], and inferred causal relationships in data management systems [82, 191, 192] can help explain query results and suggest oracles for systems that rely on data management systems [194]. Such inference can also help debug errors [278, 279, 280] by tracking and using data provenance [193].

Dynamic invariant mining, e.g., Daikon [75], can infer oracles from test executions by observing arguments' values method return values [205]. Such oracles are a kind of regression testing, ensuring only that behavior does not change during software evolution. Korat uses formal specifications of pre- and post-conditions (e.g., written by the developer or inferred by invariant mining) to generate oracles and tests [24]. By contrast, Swami, our technique of generating tests, infers oracles from the specification and neither requires source code nor an existing test suite.

CHAPTER 8

CONTRIBUTIONS AND FUTURE WORK

In this chapter, I will summarize the contributions presented in this dissertation, and present future directions of my research, some of which I intend to follow.

8.1 Contributions

While existing APR techniques can fix a large number of bugs in real-world software, most of the repairs produced are not "correct" or acceptable to the developers. This is a critical concern which prevents program repair techniques to be used in real-life software development processes. This dissertation presented multiple methods to address this problem.

We defined objective evaluation frameworks to evaluate the applicability and quality of the APR techniques and evaluated state-of-the-art repair techniques using our frameworks. We also developed and evaluated the repair quality of SOSRepair, a novel semantics-based repair technique that produces high-quality patches for real-world defects in large C programs. Further, we analyzed various factors that could affect the quality of APR. The key findings from these evaluations revealed that developer-written test suites (used in the repair process), and fault localization accuracy can significantly affect the quality of repair techniques.

This motivated us develop methods to improve developer-written test suites and automated fault localization used to guide the repair process. We improved developer tests and fault localization by extracting information from natural-language software artifacts such as specification documents and bug reports, which are not typically

used by APR techniques. Finally, we demonstrated that by using both tests and bug reports together to guide the repair process can significantly improve the quality of multiple state-of-the-art APR techniques that use fundamentally different repair approaches.

8.2 Future work

In this section, I will describe the work that remains to be done in the areas of fault localization, test generation, and program repair. Further, I will describe some of the ideas I plan to work on in the future.

8.2.1 End-To-End Automated Software Debugging

The goal of this research is to automate the end-to-end software debugging process by developing tools that can automatically synthesize or repair software requirements, design, source code, and tests from natural language software specifications. While there exists decades of research in automated FL to help developers in efficiently localizing bugs in their programs, most of these techniques are not much useful to practitioners because it is often not sufficient to just know the defective location to understand the root cause of the bug and developers end up spending the same amount of time inspecting the ranked list of suspicious statements as they would spend in understanding the bug information sources such as failing tests or bug report. On the other hand, APR research focuses mostly on fixing bugs without focusing much on understanding and effectively localizing bugs. Similar to the FL techniques, APR techniques are not much used by practitioners because developers are reluctant to review multiple patches and they find it hard to trust the correctness of patches [207]. While researchers in FL and APR are independently and actively working to improve the state-of-the-art, it is important to work together in devising end-to-end debugging techniques that are actually useful for practitioners. In this dissertation, I demonstrate

how advancements in the FL research can improve APR. A recent study [178] shows the reverse, i.e., how APR can improve FL. I plan to conduct research that synthesizes knowledge from independent yet closely related research areas.

8.2.2 Improving Software Quality Using Unstructured Text

While Swami (Chapter 4 uses a rule-based natural language processing (NLP) approach to generate tests, which limits its generalization to all kinds of natural language specifications, it can be used to synthesize large datasets for training the modern NLP techniques that can then be used to generate tests from more kinds of natural language specifications that what Swami can currently process. The modern deep-learning based NLP approaches, did not perform well for our problem because: (a) we did not have a large dataset of specifications available for a software, and (b) for a single specification there are many test cases with varying test inputs but same test oracles. These two factors posed a new challenge of training a deep-learning model on the many-to-one mappings between specifications and test cases. I plan to solve these challenges by using Swami-generated tests and the advancements made in NLP research area.

8.2.3 Automatic Repair for Hard and Important Defects

Before attempting to repair defects, developers often ask questions such as "How difficult will this defect be to fix?" and "Is the defect worth fixing?". Thus, to make program repair tools useful for practitioners, researchers need to know what kind of defects developers find worth fixing and are hard to fix. To that end, I developed a framework [199] to objectively characterize defects that are hard and important from developers' perspective and used the framework to evaluate repair tools. I found that modern repair tools fix defects that are simple yet important for developers. Researchers [166, 253] have built on my work to produce more evaluation metrics and frameworks that aim to boost the development of practical and reliable program repair

tools. I plan to work on extending the capability of repair techniques to fix defects that are hard and important for developers.

8.2.4 Debugging Tools for New Programming Paradigms

Practitioners are increasingly shifting to develop software using new programming paradigms such as machine learning, deep learning, and quantum computing techniques. As the software using these new techniques are fundamentally different from traditional ones, existing software debugging techniques (techniques to automatically localize and fix bugs, and validate correctness of produced patches) do not directly apply to such software. For example, fixing bugs in an ML-based software requires inspecting both source code and the dataset used for training the software. Methods to automatically test, detect and fix bugs in ML-based software are active research areas. I plan to develop techniques that can be used to debug such new kinds of software.

8.2.5 Software Debugging Considering Socio-Technical Aspects

Studies [129] show that not all defects are considered equally by developers. For example, before attempting to fix defects developers often ask questions such as *Is this a legitimate defect?*, *How difficult will this defect be to fix?*, and *Is the defect worth fixing?*. The answers to such questions do not only relate to the expertise of developers in the program in question but also on several factors that may affect the developers' motivation to fix defects. For example, developers may only care to fix defects in the production code to be shipped in the current release cycle, or only fix the ones that would satisfy the the compliance their team cares about. The utility of automated debugging tools can be severely affected by such socio-technical factors and thus, it is not sufficient to just improve the accuracy of tools in detecting and fixing defects. I plan to develop techniques that are cognizant of such factors and provide customized experience to developers based on the their development context.

APPENDIX A

DEFECT IMPORTANCE AND DIFFICULTY DATA

Figure A.1 describes the relevant concrete parameters for each of the bug tracking systems, project-hosting platforms, and defect benchmarks. We omit the semantics of the specific names the various systems and platforms use. This information is available from the underlying bug tracking systems and project-hosting platforms. Figure A.2 shows the mapping from concrete parameters to abstract parameters and to the five defect characteristics.

Issue tracking system	**Concrete parameters relevant to importance or difficulty**	**Other relevant concrete parameters**
Bugzilla	importance (priority and severity), target milestone, dependencies (depends on and blocks), reported, modified, time tracking (orig. est., current est., hours worked, hours left, %complete, gain, deadline), priority, components	hardware (platform and OS), keywords, personal tags
FogBugz	priority, milestones, die, subcases	areas, category
GitHub	-	labels
Google code	open, closed, blockedon, blocking, priority, reproducible, star	summary+labels
HP ALM/Quality Center	severity, closing date, detected on date, priority, reproducible, estimated fix time, view linked entities	—
IBM Rational ClearQuest	severity, priority	keywords
JIRA	component/s, votes, watchers, due, created, updated, resolved, estimate, remaining, logged, priority, severity, affects versions, fix versions	environment, labels
Mantis	reproducibility, date submitted, last update	category, profile, platform, OS, tags
Redmine	priority, updated, related issues, associated revisions, start, due date, estimated time(hours)	category
SourceForge	created, updated, priority, milestone	keywords, milestone
Trac	component, priority, milestone	keywords
Defects4J	# of files in the developer-written patch, # of lines in the developer-written patch, # of relevant tests, # of triggering tests, coverage information of test suit	—
ManyBugs	# of files in the developer-written patch, # of lines in the developer-written patch, # of positive tests, # of negative tests	developer-written patch modifications types, defect types

Figure A.1: We used grounded theory to extract from bug tracking systems, project-hosting platforms, and defect benchmarks the concrete parameters relevant to defect importance and difficulty, as well as several other parameters interesting to correlate with automated repair techniques' ability to repair the defect.

Defect characteristic	**Abstract parameter**	**Concrete parameters**
Importance	**Time to fix**: the amount of time (days) taken by developer(s) to fix a defect. This is computed as the time difference between when the issue was reported and when the issue was resolved. Depending on the issue tracking system, different concrete parameters are used to obtain these two timestamps.	reported, modified, time tracking (orig. est., current est., hours worked, hours left, %complete, gain, deadline), due, created, updated, resolved, estimate, remaining, logged, date submitted, last update, start, due date, estimated time (hours), closing date, detected on date, estimated fix time, opened, closed, milestone
	Priority: importance of fixing a defect in terms of defect priority. This is obtained using priority assigned to the defect. Different issue tracking systems use different values to denote low, normal, high, critical, blocker defects. We use a scale of 1 to 5 corresponding to these priority values (1 is the lowest priority and 5 is the highest) and map the values used by issue tracking systems to our scale. Significance is measured using the number of watchers for a defect, or the number of votes.	priority, importance (priority and severity), watchers, votes, stars
	Versions: effect of defect on different versions of a project or other project modules and components.	components, linked entities, affects versions, fix versions
Complexity	**File count**:the number of files containing non-comment, non-blank-line edits in the developer-written fix	information available from commits on issue tracking systems and helper scripts provided by Defects4J repository.
	Line count: the total number of non-comment, non-blank lines of code in the developer-written fix	information obtained using diff between buggy and fixed source code files. Helper scripts provided with Defects4J
	Reproducibility: how easy it is to reproduce the defect	reproducible, reproducibility
Test Effectiveness	**Statement coverage**: the fraction of the lines in the files edited by the developer-written patches that are executed by the test suite	provided by Defects4J framework
	Triggering test count: number of defect triggering test cases	information provided in `test.sh` script in ManyBugs and triggering_tests in Defects4J
	Relevant test count: number of test cases that execute at least one statement in at least one file edited by the developer-written patch	information provided in `test.sh` script in ManyBugs and relevant_tests in Defects4J
Independence	**Dependents**: number of defects (also with URLs to issue tracking systems) on which the fixing of a given defect depends	Dependencies (depends on and blocks), blockedon, related issues, subcases
Characteristics of the developer-written patch	**Patch characteristics**: characteristics of the developer-written patch in terms of the type of code modifications done to fix the defect	information about bug type available within the ManyBugs metadata

Figure A.2: Mapping of the concrete parameters from Figure A.1 to the eleven abstract parameters and then to the five defect characteristics.

APPENDIX B

AVAILABILITY OF APPLICABILITY DATA FOR ANNOTATING DEFECT BENCHMARKS

Figure B.1 describes information about which abstract parameters were available in different issue tracking systems used by ManyBugs and Defects4J projects and how the corresponding concrete parameters were used to annotate the defects. Figure B.2 shows the number of defects annotated for each abstract parameter using concrete parameters from bug trackers and benchmarks.

ManyBugs						
project	**issue tracking system**	**time to fix**	**priority**	**versions**	**dependents**	**reproducibility**
PHP	php bugs	difference between timestamps of "Modified" and "Submitted"	NA	number of values in "PHP Version"	NA	NA
python	python bugs	difference between timestamps of "Last changed" and "created on"	value of "Priority" scaled to our range of [1,5] as: low → 1; normal → 2; high → 3; critical → 4; deferred blocker → 5; release blocker → 5	number of versions in "Versions"	number of values in "Dependencies"	NA
gzip	mail archive	NA	NA	number of versions in "Version"	NA	NA
	Debian bugs	NA	value of "Severity" scaled to our range of [1,5] as: critical → 5; grave → 5; serious → 4; important → 3; normal → 2; minor → 1	number of versions in "Version"	NA	NA
libtiff	Bugzilla Map Tools	difference in timestamps of "Modified" and "Reported"	NA	number of versions in "Version"	value of "Depends on:"	NA
valgrind	KDE Bug tracking System	difference in timestamps of "Modified" and "Reported"	value of "Importance"	number of versions in "Version" field	information in "Dependency tree/graph"	NA
lighttpd	Redmine	NA	value of "Priority:" scaled to our range of [1,5] as: low → 1; normal → 2; high → 3; urgent → 4; immediate → 5	values in "Target version:"	NA	NA
Defects4J						
Commons Math	Apache issues	value of "Open → Resolved" field in "Transitions" tab	value of "Priority"	number of versions in "Affected versions" and "fix versions"	number of "Issue Links"	NA
Commons Lang	Apache issues	value of,"Open → Resolved" field in "Transitions" tab	value of "Priority"	number of versions in,"Affected versions" and "fix versions"	number of "Issue Links"	NA
JFreeChart	Sourceforge	difference between timestamps of "Updated" and "Created"	value of "Priority" scaled to our range of [1,5] as: 4 → 1; 5 → 2; 6 → 3,7; 8 → 4;9 → 5	NA	NA	NA
JodaTime	Github	difference between timestamps of "Created" and "Last Commit"	NA	NA	NA	NA
	Sourceforge	difference between timestamps of "Updated" and "Created"	value of "Priority" scaled to our range [1,5] as: 4 → 1; 5 → 2; 6 → 3,7; 8 → 4;9 → 5	NA	NA	NA

Figure B.1: Information about abstract parameters obtained from the issue tracking systems.

ManyBugs							
time to fix:	105	file count:	185	statement coverage:	133	dependents:	88
priority:	25	line count:	185	triggering test count:	185	patch characteristics:	185
versions:	111	reproducibility:	0	relevant test count:	185		
Defects4J							
time to fix:	203	file count:	224	statement coverage:	224	dependents:	169
priority:	191	line count:	224	triggering test count:	224	patch characteristics:	224
versions:	169	reproducibility:	0	relevant test count:	224		

Figure B.2: The number of defects annotated for each abstract parameter using the information described in Figure B.1 and data available in the ManyBugs and Defects4J benchmarks.

APPENDIX C

SOSREPAIR: EXPRESSIVE SEMANTIC SEARCH FOR REAL-WORLD PROGRAM REPAIR

Automated program repair holds the potential to significantly reduce software maintenance effort and cost. However, recent studies have shown that it often produces low-quality patches that repair some but break other functionality. We hypothesize that producing patches by replacing likely faulty regions of code with semantically-similar code fragments, and doing so at a higher level of granularity than prior approaches can better capture abstraction and the intended specification, and can improve repair quality. We create SOSRepair, an automated program repair technique that uses semantic code search to replace candidate buggy code regions with behaviorally-similar (but not identical) code written by humans. SOSRepair is the first such technique to scale to real-world defects in real-world systems. On a subset of the ManyBugs benchmark of such defects, SOSRepair produces patches for 22 (34%) of the 65 defects, including 3, 5, and 6 defects for which previous state-of-the-art techniques Angelix, Prophet, and GenProg do not, respectively. On these 22 defects, SOSRepair produces more patches (9, 41%) that pass all independent tests than the prior techniques. We demonstrate a relationship between patch granularity and the ability to produce patches that pass all independent tests. We then show that fault localization precision is a key factor in SOSRepair's success. Manually improving fault localization allows SOSRepair to patch 23 (35%) defects, of which 16 (70%) pass all independent tests. We conclude that (1) higher-granularity, semantic-based patches can improve patch quality, (2) semantic search is promising for producing high-quality real-world defect repairs, (3) research in fault localization can significantly improve

the quality of program repair techniques, and (4) semi-automated approaches in which developers suggest fix locations may produce high-quality patches.

C.1 Introduction

Automated program repair techniques (e.g., [19, 39, 42, 53, 54, 123, 142, 143, 145, 150, 174, 177, 180, 190, 239, 264, 269, 273, 296, 300]) aim to automatically produce software patches that fix defects. For example, Facebook uses two automated program repair tools, SapFix and Getafix, in their production pipeline to suggest bug fixes [183, 249]. The goal of automated program repair techniques is to take a program and a suite of tests, some of which that program passes and some of which it fails, and to produce a patch that makes the program pass all the tests in that suite. Unfortunately, these patches can repair some functionality encoded by the tests, while simultaneously breaking other, undertested functionality [252]. Thus, *quality* of the resulting patches is a critical concern. Recent results suggest that patch overfitting — patches that pass a particular set of test cases supplied to the program repair tool but fail to generalize to the desired specification — is common [252], [146, 176, 227]. The central goal of this work is to improve the ability of automated program repair to produce high-quality patches on real-world defects.

We hypothesize that producing patches by (1) replacing likely faulty regions of code with semantically-similar code fragments, and (2) doing so at a higher level of granularity than prior approaches can improve repair quality. There are two underlying reasons for this hypothesis:

1. The observation that human-written code is highly redundant [14, 35, 36, 84, 186], suggesting that, for many buggy code regions intended to implement some functionality, there exist other code fragments that seek to implement the same functionality, and at least one does so correctly.

2. Replacing code at a high level of granularity (e.g., blocks of 3–7 consecutive lines of code) corresponds to changes at a higher level of abstraction, and is thus more likely to produce patches that correctly capture the implied, unwritten specifications underlying desired behavior than low-level changes to tokens or individual lines of code.

For example, suppose a program has a bug in a loop that is intended to sort an array. First, consider another, semantically similar loop, from either the same project, or some other software project. The second loop is semantically similar to the buggy loop because, like the buggy loop, it sorts some arrays correctly. At the same time, the second loop may not be semantically identical to the buggy loop, especially on the inputs that the buggy loop mishandles. We may not know a priori if the second, similar loop is correct. However, sorting is a commonly implemented subroutine. If we try to replace the buggy code with several such similar loops, at least one is likely to correctly sort arrays, allowing the program to pass the test cases it previously failed. In fact, the high redundancy present in software source code suggests such commonly implemented subroutines are frequent [14, 35, 36, 84]. Second, we posit that replacing the entire loop with a similar one is more likely to correctly encode sorting than what could be achieved by replacing a + with a -, or inserting a single line of code in the middle of a loop.

Our earlier work on semantic-search-based repair [122] presented one instance that demonstrated that higher-granularity, semantic-based changes can, in fact, improve quality. On short, student-written programs, on average, SearchRepair patches passed 97.3% of independent tests not used during patch construction. Meanwhile, the relatively lower-granularity patches produced by GenProg [150], TrpAutoRepair [224], and AE [282] passed 68.7%, 72.1%, and 64.2%, respectively [122]. Unfortunately, as we describe next, SearchRepair cannot apply to large, real-world programs.

This chapter presents SOSRepair, a novel technique that uses input-output-based semantic code search to automatically find and contextualize patches to fix real-world defects. SOSRepair locates likely buggy code regions, identifies similarly-behaving fragments of human-written code, and then changes the context of those fragments to fit the buggy context and replace the buggy code. Semantic code search techniques [232, 259, 260, 261] find code based on a specification of desired behavior. For example, given a set of input-output pairs, semantic code search looks for code fragments that produce those outputs on those inputs. Constraint-based semantic search [259, 260, 261] can search for partial, non-executable code snippets. It is a good fit for automated program repair because it supports searching for code fragments that show the same behavior as a buggy region on initially passing tests, while looking for one that passes previously-failing tests as well.

While SOSRepair builds on the ideas from SearchRepair [122], to make SOSRepair apply, at scale, to real-world defects, we redesigned the entire approach and developed a conceptually novel method for performing semantic code search. The largest program SearchRepair has repaired is a 24-line C program written by a beginner programmer to find the median of three integers [122]. By contrast, SOSRepair patches defects made by professional developers in real-world, multi-million-line C projects. Since SearchRepair cannot run on these real-world defects, we show that SOSRepair outperforms SearchRepair on the IntroClass benchmark of small programs.

We evaluate SOSRepair on 65 real-world defects of 7 large open-source C projects from the ManyBugs benchmark [149]. SOSRepair produces patches for 22 defects, including 1 that has not been patched by prior techniques (Angelix [190], Prophet [177], and GenProg [150]). We evaluate patch quality using held-out independent test suites [252]. Of the 22 defects for which SOSRepair produces patches, 9 (41%) pass all the held-out tests, which is more than the prior techniques produce for these defects. On small C programs in the IntroClass benchmark [149], SOSRepair

generates 346 patches, more than SearchRepair [122], GenProg [150], AE [284], and TrpAutoRepair [224]. Of those patches, 239 pass all held-out tests, again, more than the prior techniques.

To make SOSRepair possible, we make five major contributions to both semantic code search and program repair:

1. **A more-scalable semantic search query encoding.** We develop a novel, efficient, general mechanism for encoding semantic search queries for program repair, inspired by input-output component-based program synthesis [106]. This encoding efficiently maps the candidate fix code to the buggy context using a single query over an arbitrary number of tests. By contrast, SearchRepair [122] required multiple queries to cover all test profiles and failed to scale to large code databases or queries covering many possible permutations of variable mappings. Our new encoding approach provides a significant speedup over the prior approach, and we show that the speedup grows with query complexity.

2. **Expressive encoding capturing real-world program behavior.** To apply semantic search to real-world programs, we extend the state-of-the-art constraint encoding mechanism to handle real-world C language constructs and behavior, including structs, pointers, multiple output variable assignments, console output, loops, and library calls.

3. **Search for patches that insert and delete code.** Prior semantic-search-based repair could only *replace* buggy code with candidate fix code to affect repairs [122]. We extend the search technique to encode deletion and insertion.

4. **Automated, iterative search query refinement encoding negative behavior.** We extend the semantic search approach to include negative behavioral examples, making use of that additional information to refine queries. We also propose a novel, iterative, and counter-example-guided search-query

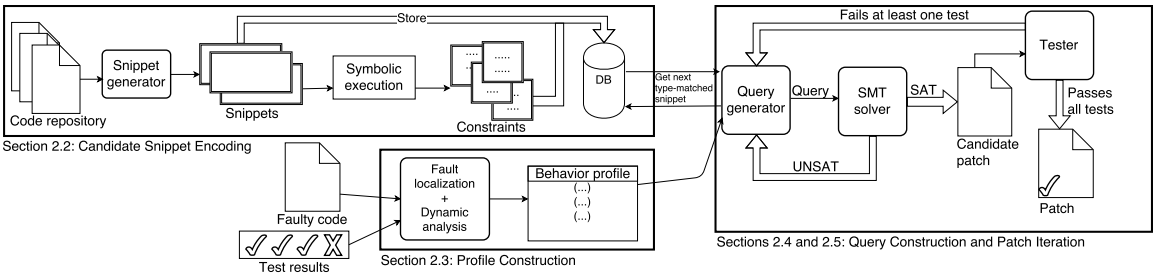


Figure C.1: Overview of the SOSRepair approach.

refinement approach to repair buggy regions that are not covered by the passing test cases. When our approach encounters candidate fix code that fails to repair the program, it generates new undesired behavior constraints from the new failing executions and refines the search query, reducing the search space. This improves on prior work, which could not repair buggy regions that no passing test cases execute [122].

5. **Evaluation and open-source implementation.** We implement and release SOSRepair (`https://github.com/squaresLab/SOSRepair`), which reifies the above mechanisms. We evaluate SOSRepair on the ManyBugs benchmark [149] commonly used in the assessment of automatic patch generation tools (e.g., [177, 190, 224, 282]). These programs are four orders of magnitude larger than the benchmarks previously used to evaluate semantic-search-based repair [122]. We show that, as compared to previous techniques applied to these benchmarks (Angelix [190], Prophet [177], and GenProg [150]), SOSRepair patches one defect none of those techniques patch, and produces patches of comparable quality to those techniques. We measure quality objectively, using independent test suites held out from patch generation [252]. We therefore also release independently-generated held-out test suites we use to evaluate SOSRepair. (`https://github.com/squaresLab/SOSRepair-Replication-Package`)

Based on our experiments, we hypothesize that fault localization's imprecision on real-world defects hampers SOSRepair. We create SOSRepair$^{\oplus}$, a semi-automated version of SOSRepair that is manually given the code location in which a human would repair the defect. SOSRepair$^{\oplus}$ produces patches for 23 defects. For 16 (70%) of the defects, the produced patches pass all independent tests. Thus, SOSRepair$^{\oplus}$ is able to produce high-quality patches for twice the number of defects than SOSRepair produces (16 versus 9). This suggests that semantic code search holds promise for producing high-quality repairs for real-world defects, perhaps in a semi-automated setting in which developers suggest code locations to attempt fixing. Moreover, advances in automated fault localization can directly improve automated repair quality.

To directly test the hypothesis that patch granularity affects the ability to produce high-quality patches, we alter the granularity of code SOSRepair can replace when producing patches, allowing for replacements of 1 to 3 lines, 3 to 7 lines, or 6 to 9 lines of code. On the IntroClass benchmark, using the 3–7-line granularity results in statistically significantly more patches (346 for 3–7-, 188 for 1–3-, and 211 for 6–9-line granularities) and statistically significantly more patches that pass all the held-out tests (239 for 3–7-, 120 for 1–3-, and 125 for 6–9-line granularities).

The rest of this chapter is organized as follows. Section C.2 describes the SOSRepair approach and Section C.3 our implementation of that approach and Section C.4 evaluates SOSRepair, and Section C.5 summarizes our contributions.

C.2 The SOSRepair Approach

Figure C.1 overviews the SOSRepair approach. Given a program and a set of test cases capturing correct and buggy behavior, SOSRepair generates patches by searching over a database of snippets of human-written code. Unlike keyword or syntactic search (familiar to users of standard search engines), *semantic* search looks for code based on a specification of desired and undesired behavior. SOSRepair uses test cases to

construct a behavioral profile of a potentially buggy code region. SOSRepair then searches over a database of snippets for one that implements the inferred desired behavior, adapts a matching snippet to the buggy code's context, and patches the program by replacing the buggy region with patch code, inserting patch code, or deleting the buggy region. Finally, SOSRepair validates the patched program by executing its test cases.

We first describe an illustrative example and define key concepts (Section C.2.1). We then detail SOSRepair's approach that (1) uses symbolic execution to produce static behavioral approximations of a set of candidate bug repair snippets (Section C.2.2), (2) constructs a dynamic profile of potentially-buggy code regions, which serve as inferred input-output specifications of desired behavior (Section C.2.3), (3) constructs an SMT query to identify candidate semantic repairs to be transformed into patches and validated (Section C.2.4), and (4) iteratively attempts to produce a patch until timeout occurs (Section C.2.5). This section focuses on the conceptual approach; Section C.3 will describe implementation details.

C.2.1 Illustrative example and definitions

Consider the example patched code in Figure C.2 (top), which we adapt (with minor edits for clarity and exposition) from `php` interpreter bug issue #60455, concerning a bug in the `streams` API.[1] Bug #60455 reports that `streams` mishandles files when the EOF character is on its own line. The fixing commit message elaborates: "stream_get_line misbehaves if EOF is not detected together with the last read." The change forces the loop to continue such that the last EOF character is consumed. The logic that the developer used to fix this bug is not unique to the `stream_get_record` function; indeed, very similar code appears in the `php date` module (bottom of Figure C.2). This

[1] `https://bugs.php.net/bug.php?id=60455`

```
1       // len holds current position in stream
2       while (len < maxlen) {
3       php_stream_fill_read_buffer(stream,
4       len + MIN(maxlen- len chunk_size));
5       just_read =
6       (stream->writepos - stream->readpos)-len;
7       - if (just_read < toread) {
8       + if (just_read == 0) {
9       break;
10      } else {
11      len = len + just_read;
12      }
13      }
```

```
1       if (bufflen > 0)
2       mylen += bufflen;
3       else break;
```

Figure C.2: Top: Example code, based on `php` bug # 60455, in function `stream_get_record`. The developer patch modifies the condition on line 7, shown on line 8. Bottom: A snippet appearing in the `php` date module, implementing the same functionality as the developer patch (note that `just_read` is never negative in this code), with different variable names.

is not unusual: there exists considerable redundancy within and across open-source repositories [14, 84, 102, 271].

Let $\mathcal{F}$ refer to a code snippet of 3–7 lines of C code. $\mathcal{F}$ can correspond to either the buggy region to be replaced or a snippet to be inserted as a repair. In our example bug, a candidate buggy to-be-replaced region is lines 7–11 in top of Figure C.2; the snippet in the bottom of Figure C.2 could serve as a repair snippet. We focus on snippets of size 3–7 lines of code because patches at a granularity level greater than single-expression, -statement, or -line may be more likely to capture developer intuition, producing more-correct patches [122], but code redundancy drops off sharply beyond seven lines [84, 102]. We also verify these findings by conducting experiments that use code snippets of varying sizes (Section C.4.3).

$\mathcal{F}$'s *input* variables $\vec{X_f}$ are those whose values can ever be *used* (in the classic dataflow sense, either in a computation, assignment, or predicate, or as an argument

to a function call); $\mathcal{F}$'s *output* variables $\vec{R_f}$ are those whose value may be *defined* with a definition that is not *killed* by the end of the snippet. In the buggy region of Figure C.2, $\vec{X_f}$ is {`just_read`, `toread`, `len`}; $\vec{R_f}$ is {`len`}. $\vec{R_f}$ may be of arbitrary size, and $\vec{X_f}$ and $\vec{R_f}$ are not necessarily disjoint, as in our example. $\vec{\mathcal{V}_f}$ is the set of all variables of interest in $\mathcal{F}$: $\vec{\mathcal{V}_f} = \vec{X_f} \cup \vec{R_f}$.

To motivate a precise delineation between variable uses and definitions, consider a concrete example that demonstrates correct behavior for the buggy code in Figure C.2: if `just_read` = 5 and `len` = 10 after line 6, at line 12, it should be the case that `just_read` = 5 and `len` = 15. A naive, constraint-based expression of this desired behavior, e.g., (`just_read` = 5) $\wedge$ (`len` = 10) $\wedge$ (`just_read` = 5) $\wedge$ (`len` = 15) is unsatisfiable, because of the conflicting constraints on `len`.

For the purposes of this explanation, we first address the issue by defining a static variable renaming transformation over snippets. Let $U_f(x)$ return all uses of a variable x in $\mathcal{F}$ and $D_f(x)$ return all definitions of x in $\mathcal{F}$ that are not killed. We transform arbitrary $\mathcal{F}$ to enforce separation between inputs and outputs as follows:

$$\mathcal{F}' = F[U_f(x)/x_i] \;\; s.t. x \in V_f, x_i \in X_{in}, x_i \; \textit{fresh}$$
$$\mathcal{F}_t = F'[D_f(x)/x_i] \; s.t. x \in R_f, x_i \in X_{out}, x_i \; \textit{fresh}$$

All output variables are, by definition, treated also as inputs, and we choose fresh names as necessary. X_{in} and X_{out} refer to the sets of newly-introduced variables.

C.2.2 Candidate snippet encoding

In an offline pre-processing step, we prepare a database of candidate repair snippets of 3–7 lines of C code. This code can be from any source, including the same project, its previous versions, or open-source repositories. A naive lexical approach to dividing code into line-based snippets generates many implausible and syntactically invalid snippets, such as by crossing block boundaries (e.g., lines 10–12 in the top of Figure C.2).

Instead, we identify candidate repair snippets from C blocks taken from the code's abstract syntax tree (AST). Blocks of length 3–7 lines are treated as a single snippet. Blocks of length less than 3 lines are grouped with adjacent blocks. We transform all snippets $\mathcal{F}$ into $\mathcal{F}_t$ (Section C.2.1). In addition to the code itself (pre- and post-transformation) and the file in which it appears, the database stores two types of information per snippet:

1. **Variable names and types.** Patches are constructed at the AST level, and are thus always syntactically valid. However, they can still lead to compilation errors if they reference out-of-scope variable names, user-defined types, or called functions. We thus identify and store names of user-defined structs and called functions (including the file in which they are defined). We additionally store all variable names from the *original* snippet $\mathcal{F}$ ($\vec{\mathcal{V}_f}$, $\vec{X_f}$, $\vec{R_f}$), as well as their corresponding renamed versions in $\mathcal{F}_t$ (X_{in} and X_{out}).

2. **Static path constraints.** We symbolically execute [33, 125] $\mathcal{F}_t$ to produce a symbolic formula that statically overapproximates its behavior, described as constraints over snippet input and outputs. For example, the fix snippet in Figure C.2 can be described as:

$$((\texttt{bufflen}_{in} > 0) \wedge (\texttt{mylen}_{out} = \texttt{mylen}_{in} + \texttt{bufflen}_{in}))$$
$$\vee(\neg(\texttt{bufflen}_{in} > 0) \wedge (\texttt{mylen}_{out} = \texttt{mylen}_{in}))$$

   We query an SMT solver to determine whether such constraints match desired inputs and outputs.

The one-time cost of database construction is amortized across many repair efforts.

C.2.3 Profile construction

SOSRepair uses spectrum-based FL (SBFL) [117] to identify candidate buggy code regions. SBFL uses test cases to rank program entities (e.g., lines) by suspiciousness.

We expand single lines identified by SBFL to the enclosing AST block. Candidate buggy regions may be smaller than 3 lines if no region of fewer than 7 lines can be created by combining adjacent blocks.

Given a candidate buggy region $\mathcal{F}$, SOSRepair constructs a *dynamic profile* of its behavior on passing and failing tests. Note that the profile varies by the type of repair, and that SOSRepair can either *delete* the buggy region; *replace* it with a candidate repair snippet; or *insert* a piece of code immediately before it. We discuss how SOSRepair iterates over and chooses between repair strategies in Section C.2.5. Here, we describe profile generation for replacement and insertion (the profile is not necessary for deletion).

SOSRepair first statically substitutes $\mathcal{F}_t$ for $\mathcal{F}$ in the buggy program, declaring fresh variables X_{in} and X_{out}. SOSRepair then executes the program on the tests, capturing the values of all local variables before and after the region on all covering test cases. (For simplicity and without loss of generality, this explanation assumes that all test executions cover all input and output variables.) Let T_p be the set of all initially passing tests that cover $\mathcal{F}_t$ and T_n the set of all initially failing tests that do so. If t is a test case covering $\mathcal{F}_t$, let $valIn(t, x)$ be the observed dynamic value of x on test case t before $\mathcal{F}_t$ is executed and $valOut(t, x)$ its dynamic value afterwards. We index each observed value of each variable of interest x by the test execution on which the value is observed, denoted x^t. This allows us to specify desired behavior based on multiple test executions or behavioral examples at once. To illustrate, assume a second passing execution of the buggy region in Figure C.2 on which `len` is 15 on line 6 and 25 on line 12 (ignoring `just_read` for brevity). $\Big((\texttt{len}_{in} = 10) \wedge (\texttt{len}_{out} = 15)\Big) \wedge \Big((\texttt{len}_{in} = 15) \wedge (\texttt{len}_{out} = 25)\Big)\Big)$ is trivially unsatisfiable; $\Big((\texttt{len}_{in}{}^1 = 10) \wedge (\texttt{len}_{out}{}^1 = 15)\Big) \wedge \Big((\texttt{len}_{in}{}^2 = 15) \wedge (\texttt{len}_{out}{}^2 = 25)\Big)\Big)$, which indexes the values by the tests on which they were observed, is not. The dynamic profile is then defined as follows:

$$P := P_{in} \wedge P^p_{out} \wedge P^n_{out}$$

P_{in} encodes bindings of variables to values on entry to the candidate buggy region on all test cases; P^p_{out} enforces the desired behavior of output variables to match that observed on initially passing test cases; P^n_{out} enforces that the output variables should *not* match to those observed on initially failing test cases. P_{in} is the same for both replacement and insertion profiles:

$$P_{in} := \bigwedge_{t \in T_p \cup T_n} \bigwedge_{x_i \in X_{in}} x_i^t = valIn(t, x_i)$$

P_{out} combines constraints derived from both passing and failing executions, or $P^p_{out} \wedge P^n_{out}$. For replacement queries:

$$\begin{aligned} P^p_{out} &:= \bigwedge_{t \in T_p} \bigwedge_{x_i \in X_{out}} x_i^t = valOut(t, x_i) \\ P^n_{out} &:= \bigwedge_{t \in T_n} \neg(\bigwedge_{x_i \in X_{out}} x_i^t = valOut(t, x_i)) \end{aligned}$$

For insertion queries, the output profile specifies that the correct code should simply preserve observed passing behavior while making some observable change to initially failing behavior:

$$\begin{aligned} P^p_{out} &:= \bigwedge_{t \in T_p} \bigwedge_{x_i \in X_{out}} x_i^t = valIn(t, x_i) \\ P^n_{out} &:= \bigwedge_{t \in T_n} \neg(\bigwedge_{x_i \in X_{out}} x_i^t = valIn(t, x_i)) \end{aligned}$$

Note that we neither know, nor specify, the correct value for these variables on such failing tests, and do not require annotations or developer interaction to provide them such that they may be inferred.

C.2.4 Query construction

Assume candidate buggy region $\mathcal{C}$ (a *context* snippet), candidate repair snippet $\mathcal{S}$, and corresponding input variables, output variables, etc. (as described in Section C.2.1). Our goal is to determine whether the repair code $\mathcal{S}$ can be used to edit the buggy code, such that doing so will possibly address the buggy behavior without breaking previously-correct behavior. This task is complicated by the fact that candidate repair snippets may implement the desired behavior, but use the wrong variable names for the buggy context (such as in our example in Figure C.2). We solve this problem by constructing a single SMT query for each pair of $\mathcal{C}$, $\mathcal{S}$, that identifies whether a mapping exists between their variables ($\vec{\mathcal{V}}_c$ and $\vec{\mathcal{V}}_s$) such that the resulting patched code ($\mathcal{S}$ either substituted for or inserted before $\mathcal{C}$) satisfies all the profile constraints P. An important property of this query is that, if satisfiable, the satisfying model provides a variable mapping that can be used to rename $\mathcal{S}$ to fit the buggy context.

The repair search query is thus comprised of three constraint sets: (1) mapping components ψ_{map} and ψ_{conn}, which enforce a valid and meaningful mapping between variables in the candidate repair snippet and ones in the buggy context, (2) functionality component ϕ_{func}, which statically captures the behavior of the candidate repair snippet, and (3) the specification of desired behavior, captured in a dynamic profile P (Section C.2.3). We now detail the mapping and functionality components, as well as how patches are constructed and validated based on satisfiable semantic search SMT queries.

C.2.4.1 Mapping component

Our approach to encoding semantic search queries for APR takes inspiration from SMT-based input-output-guided component-based program synthesis [106]. The original synthesis goal is to connect a set of components to construct a function f that satisfies a set of input-output pairs $\langle \alpha_i, \beta_i \rangle$ (such that $\forall i, f(\alpha_i) = \beta_i$). This

is accomplished by introducing a set of *location variables*, one for each possible component and function input and output variable, that define the order of and connection between components. Programs are synthesized by constructing an SMT query that constrains location variables so that they describe a well-formed program with the desired behavior on the given inputs/outputs. If the query is satisfiable, the satisfying model assigns integers to locations and can be used to construct the desired function. See the prior work by Jha et al. for full details [106].

Mapping queries for replacement. We extend the location mechanism to map between the variables used in a candidate repair snippet and those available in the buggy context. We first describe how mapping works for replacement queries, and then the differences required for insertion. We define a set of *locations* as:

$$L = \{l_x | x \in \vec{\mathcal{V}_c} \cup \vec{\mathcal{V}_s}\}$$

The query must constrain locations so that a satisfying assignment tells SOSRepair how to suitably rename variables in $\mathcal{S}$ such that a patch compiles and enforces desired behavior. The variable mapping must be valid: Each variable in $\mathcal{S}$ must uniquely map to some variable in $\mathcal{C}$ (but not vice versa; not all context snippet variables need map to a repair snippet variable). The ψ_{map} constraints therefore define an injective mapping from $\vec{\mathcal{V}_s}$ to $\vec{\mathcal{V}_c}$:

$$\begin{aligned} \psi_{map} \quad &:= \quad \Big(\bigwedge_{x \in \vec{\mathcal{V}_c} \cup \vec{\mathcal{V}_s}} 1 \leq l_x \leq |\vec{\mathcal{V}_c}| \Big) \\ &\quad\quad \wedge \ \mathit{distinct}(L, \vec{\mathcal{V}_c}) \wedge \ \mathit{distinct}(L, \vec{\mathcal{V}_s}) \\ \mathit{distinct}(L, \vec{V}) \quad &:= \quad \bigwedge_{x,y \in \vec{V}, x \not\equiv y} l_x \neq l_y \end{aligned}$$

This exposition ignores variable types for simplicity; in practice, we encode them such that matched variables have the same types via constraints on valid locations.

Next, ψ_{conn} establishes the connection between location values and variable values as well as between input and output variables $\vec{X_s}$, $\vec{R_s}$ and their freshly-renamed versions in

X_{in} and X_{out} across all covering test executions $t \in T_p \cup T_n$. This is important because although the introduced variables eliminate the problem of trivially unsatisfiable constraints over variables used as both inputs and outputs, naive constraints over the fresh variables — e.g., $(\texttt{len}_{in}{}^1 = 10) \wedge (\texttt{len}_{out}{}^1 = 15)$ — are instead trivially satisfiable. Thus:

$$\begin{aligned}
\psi_{conn} &:= \psi_{out} \wedge \psi_{in} \\
\psi_{out} &:= \bigwedge_{x \in X^C_{out}, y \in X^S_{out}} l_x = l_y \implies \\
&\qquad \Big(\bigwedge_{t=1}^{|T_p \cup T_n|} x^t_{in} = y^t_{in} \wedge x^t_{out} = y^t_{out} \Big) \\
\psi_{in} &:= \bigwedge_{x \in X^C_{in}, y \in X^S_{in}} l_x = l_y \implies \Big(\bigwedge_{t=1}^{|T_p \cup T_n|} x^t_{in} = y^t_{in} \Big)
\end{aligned}$$

Where X^C_{in} and X^S_{in} refer to the variables in the context and repair snippet respectively and x_{in} refers to the fresh renamed version of variable x, stored in X_{in} (and similarly for output variables).

Insertion. Instead of drawing $\vec{\mathcal{V}}_c$ from the replacement region (a heuristic design choice to enable scalability), insertion queries define $\vec{\mathcal{V}}_c$ as the set of local variables live after the candidate insertion point. They otherwise are encoded as above.

C.2.4.2 Functionality component

ϕ_{func} uses the path constraints describing the candidate repair snippet $\mathcal{S}$ such that the query tests whether $\mathcal{S}$ satisfies the constraints on the desired behavior described by the profile constraints P. The only complexity is that we must copy the symbolic formula to query over multiple simultaneous test executions. Let φ_c be the path constraints from symbolic execution. $\varphi_c(i)$ is a copy of φ_c where all variables $x_{in} \in X^S_{in}$ and $x_{out} \in X^S_{out}$ are syntactically replaced with indexed versions of themselves (e.g., x^i_{in} for x_{in}). Then:

$$\phi_{func} := \bigwedge_{i=1}^{|T_p \cup T_n|} \varphi_c(i)$$

ϕ_{func} is the same for replacement and insertion queries.

C.2.4.3 Patch construction and validation

The repair query conjoins the above-described constraints:

$$\psi_{map} \wedge \psi_{conn} \wedge \phi_{func} \wedge P$$

Given $\mathcal{S}$ and $\mathcal{C}$ for which a satisfiable repair query has been constructed, the satisfying model assigns values to locations in L and defines a valid mapping between variables in the *original* snippets S and C (rather than their transformed versions). This mapping is used to rename variables in S and integrate it into the buggy context. For *replacement* edits, the renamed snippet replaces the buggy region wholesale; for insertions, the renamed snippet is inserted immediately before the buggy region. It is possible for the semantic search to return satisfying snippets that do not repair the bug when executed, if either the snippet fails to address the bug correctly, or if the symbolic execution is too imprecise in its description of snippet behavior. Thus, SOSRepair validates patches by running the patched program on the provided test cases, reporting the patch as a fix if all test cases pass.

C.2.5 Patch iteration

Traversal. SOSRepair iterates over candidate buggy regions and candidate repair strategies, dynamically testing all snippets whose repair query is satisfiable. SOSRepair is parameterized by a fault localization strategy, which returns a weighted list of candidate buggy lines. Such strategies can be imprecise, especially in the absence of high-coverage test suites [258]. To avoid getting stuck trying many patches in the wrong location, SOSRepair traverses candidate buggy regions using breadth-first search. First, it tries deletion at every region. Deletion is necessary to repair certain defects [318], though it can also lead to low-quality patches [227]. However, simply disallowing deletion does not solve the quality problem: even repair techniques that do not formally support deletion can do so by synthesizing tautological `if`

1: **procedure** REFINEPROFILE($program, Tests, X_{out}$)
2: $constraints \leftarrow \emptyset$
3: **for all** $t \in Tests$ **do** ▷ all tests $t \in Tests$ failed
4: $c \leftarrow \neg(\bigwedge_{x \in X_{out}} x^t = valOut(t, x, program))$
5: $constraints \leftarrow constraints \cup c$
6: **return** $constraints$

Figure C.3: Incremental, counter-example profile refinement. REFINEPROFILE receives a $program$ with the candidate snippet incorporated, a set of $Tests$ that fail on $program$, and the set of output variables X_{out}. It computes new $constraints$ to refine the profile by excluding the observed behavior. $valOut(t, x_i, program)$ returns the output value of variable x_i when test t is executed on $program$.

conditions [175, 190]. Similarly, SOSRepair can replace a buggy region with a snippet with no effect. Because patches that effectively delete are likely less maintainable and straightforward than those that simply delete, if a patch deletes functionality, it is better to do so explicitly. Thus, SOSRepair tries deleting the candidate buggy region first by replacing it with an empty candidate snippet whose only constraint is *TRUE*. We envision future improvements to SOSRepair that can create and compare multiple patches per region, preferring those that maintain the most functionality. Next, SOSRepair attempts to replace regions with identified fix code, in order of ranked suspiciousness; finally, SOSRepair tries to repair regions by inserting code immediately before them. We favor replacement over insertion because the queries are more constrained. SOSRepair can be configured with various database traversal strategies, such as trying snippets from the same file as the buggy region first, as well as trying up to N returned matching snippets per edit type per region. SOSRepair then cycles through buggy regions and matched snippets N-wise, before moving to the next edit type.

Profile refinement. Initially-passing test cases partially specify the expected behavior of a buggy code region, thus constraining which candidate snippets quality to be returned by the search. Initially-failing test cases only specify what the behavior *should not* be (e.g., "given input 2, the output should not be 4"). This is significantly

less useful in distinguishing between candidate snippets. Previous work in semantic search-based repair disregarded the negative example behavior in generating dynamic profiles [122]. Such an approach might be suitable for small programs with high-coverage test suites. Unfortunately, in real-world programs, buggy regions may only be executed by failing test cases [258]. We observed this behavior in our evaluation on real-world defects.

To address this problem, other tools, such as Angelix [190], require manual specification of the correct values of variables for negative test cases. By contrast, we address this problem in SOSRepair via a novel *incremental, counter-example-guided profile* refinement for candidate regions that do not have passing executions. Given an initial profile derived from failing test cases (e.g., "given input 2, the output should not be 4"), SOSRepair tries a single candidate replacement snippet $\mathcal{S}$. If unsuccessful, SOSRepair adds the newly discovered unacceptable behavior to the profile (e.g., "given input 2, the output should not be 6"). Figure C.3 details the algorithm for this refinement process. Whenever SOSRepair tries a snippet and observes that all tests fail, it adds one new negative-behavior constraint to the constraint profile for each failing test. Each constraint is the negation of the observed behavior. For example, if SOSRepair observes that test t fails, it computes its output variable values (e.g., $x_1 = 3$, $x_2 = 4$) and adds the constraint $\neg\left((x_1^t = 3) \wedge (x_2^t = 4)\right)$ to the profile, which specifies that the incorrect observed behavior should not take place. Thus, SOSRepair gradually builds a profile based on negative tests without requiring manual effort. SOSRepair continues on trying replacement snippets with queries that are iteratively improved throughout the repair process. Although this is slower than starting with passing test cases, it allows SOSRepair to patch more defects.

C.3 The SOSRepair Implementation

We implement SOSRepair using KLEE [33], Z3 [55], and *clang* [44] infrastructure; the latter provides parsing, name and type resolution, and rewriting facilities, among others. Section C.3.1 describes the details of our implementation. Section C.3.2 summarizes the steps we took to release our implementation and data, and to make our experiments reproducible.

C.3.1 SOSRepair implementation design choices

In implementing SOSRepair, we made a series of design decisions, which we now describe.

Snippet database. SOSRepair uses the symbolic execution engine in KLEE [33] to statically encode snippets. SOSRepair uses KLEE's built-in support for loops, using a two-second timeout; KLEE iterates over the loop as many times as possible in the allocated time. We encode user-defined struct types by treating them as arrays of bytes (as KLEE does). SOSRepair further inherits KLEE's built-in mechanisms for handling internal (GNU C) function calls. As KLEE does not symbolically execute external (non GNU C) function calls, SOSRepair makes no assumptions about such functions' side-effects. SOSRepair instead makes a new symbolic variable for each of the arguments and output, which frees these variables from previously generated constraints. These features substantially expand the expressive power of the considered repair code over previous semantic search-based repair. We do sacrifice soundness in the interest of expressiveness by casting floating point variables to integers (this is acceptable because unsoundness can be caught in testing). This still precludes the encoding of snippets that include floating point constants, but future SOSRepair versions can take advantage of KLEE's recently added floating point support.

Overall, we encode snippets by embedding them in a small function, called from `main`, and defining their input variables as symbolic (using `klee_make_symbolic`). We

use KLEE off-the-shelf to generate constraints for the snippet-wrapping function, using KLEE's renaming facilities to transform $\mathcal{F}$ into $\mathcal{F}_t$ for snippet encoding. KLEE generates constraints for nearly all compilable snippets. Exceptions are very rare, e.g., KLEE will not generate constraints for code containing function pointers. However, KLEE will sometimes conservatively summarize snippets with single *TRUE* constraints in cases where it can technically reason about code but is still insufficiently expressive to fully capture its semantics.

Console output. Real-world programs often print meaningful output. Thus, modeling console output in semantic search increases SOSRepair applicability. We thus define a symbolic character array to represent console output in candidate repair snippets. Because symbolic arrays must be of known size, we only model the first 20 characters of output. We transform calls to `printf` and `fprintf` to call `sprintf` with the same arguments. KLEE handles these standard functions natively. We track console output in the profile by logging the start and end of the buggy candidate region, considering anything printed between the log statements as meaningful.

Profile construction. For consistency with prior work [122], we use Tarantula [117] to rank suspicious source lines. We leave the exploration of other fault localization mechanisms to future work. To focus our study on SOSRepair efficacy (rather than efficiency, an orthogonal concern), we assume the provision of one buggy method to consider for repair, and then apply SBFL to rank lines in the method. Given such a ranked list, SOSRepair expands the identified lines to surrounding regions of 3–7 lines of code, as in the snippet encoding step. The size of the region is selected by conducting an initial experiment on small programs presented in Section C.4.3. SOSRepair attempts to repair each corresponding buggy region in rank order, skipping lines that have been subsumed into previously-identified and attempted buggy regions.

Queries and iteration. Z3 [55] can natively handle integers, booleans, reals, bit vectors, and several other common data types, such as arrays and pairs. To determine

whether a candidate struct type is in scope, we match struct names syntactically. For our experiments, we construct snippet databases from the rest of the program under repair, pre-fix, which supports struct matching. Additionally, programs are locally redundant [271], and developers are more often right than not [74], and thus we hypothesize that a defect may be fixable via code elsewhere in the same program. However, this may be unnecessarily restrictive for more broadly-constructed databases. We leave a more flexible matching of struct types to future work. SOSRepair is configured by default to try repair snippets from the same file as a buggy region first, for all candidate considered regions; then the same module; then the same project.

C.3.2 Open-source release and reproducibility

To support the reproduction of our results and help researchers build on our work, we publicly release our implementation: `https://github.com/squaresLab/SOSRepair`. We also release a replication package that includes all patches our techniques found on the ManyBugs benchmark and the necessary scripts to rerun the experiment discussed in Section C.4.4, and all independently generated tests discussed in Section C.4.1.2:
`https://github.com/squaresLab/SOSRepair-Replication-Package`.

Our implementation includes Docker containers and scripts for reproducing the evaluation results described next in Section C.4. The containers and scripts use BugZoo [270], a decentralized platform for reproducing and interacting with software bugs. These scripts both generate snippet databases (which our release excludes due to size) and execute SOSRepair.

SOSRepair uses randomness to make two choices during its execution: the order in which to consider equally suspicious regions returned by SOSRepair's fault localization, and the order in which to consider potential snippets returned by the SMT solver that satisfy all the query constraints. SOSRepair's configuration includes a random seed

that controls this randomness, making executions deterministic. However, there remain two sources of nondeterminism that SOSRepair cannot control. First, SOSRepair sets a time limit on KLEE's execution on each code snippet (recall Section C.3.1). Due to CPU load and other factors, in each invocation, KLEE may be able to execute the code a different number of times in the time limit, and thus generate different constraints. Second, if a code snippet contains uninitialized variables, those variables' values depend on the memory state. Because memory state may differ between executions, SOSRepair may generate different profiles on different executions. As a result of these two sources of nondeterminism, SOSRepair's results may vary between executions. However, in our experiments, we did not observe this nondeterminism affect SOSRepair's ability to find a patch, only its search space and execution time.

C.4 Evaluation

This section evaluates SOSRepair, answering several research questions. The nature of each research question informs the appropriate dataset used in its answering, as we describe in the context of our experimental methodology (Section C.4.1). We begin by using IntroClass [149], a large dataset of small, well-tested programs, to conduct controlled evaluations of:

- Comparison with prior work: How does SOSRepair perform when compared to SearchRepair [122], the prior semantic-based repair approach (Section C.4.2)?
- Tuning: What granularity level is best for the purposes of finding high-quality repairs (Section C.4.3)?

Next, in Section C.4.4, we address our central experimental concern by evaluating SOSRepair on real-world defects taken from the ManyBugs benchmark [149], by addressing:

- Expressiveness: How expressive and applicable is SOSRepair in terms of the number and uniqueness of defects it can repair?
- Quality: What is the quality and effectiveness of patches produced by SOSRepair?
- The role of fault localization: What are the limitations and bottlenecks of SOSRepair's performance?

Section C.4.5 discusses informative real-world example patches produced by SOSRepair.

Finally, we isolate and evaluate two key SOSRepair features:

- Performance improvements: How much performance improvements does the novel query encoding approach of SOSRepair afford (Section C.4.6)?
- Profile refinement: How much is the search space reduced by the negative profile refinement approach (Section C.4.7)?

Finally, we discuss threats to the validity of our experiments and SOSRepair's limitations in Section C.4.8.

C.4.1 Methodology

We use two datasets to answer the research questions outlined above. SOSRepair aims to scale semantic search repair to defects in large, real-world programs. However, such programs are not suitable for most controlled large-scaled evaluations, necessary for, e.g., feature tuning. Additionally, real-world programs preclude a comparison to previous work that does not scale to handle them. For such questions, we consider the IntroClass benchmark [149] (Section C.4.1.1). However, where possible, and particularly in our core experiments, we evaluate SOSRepair on defects from large, real-world programs taken from the ManyBugs [149] benchmark (Section C.4.1.2).

We run all experiments on a server running Ubuntu 16.04 LTS, consisting of 16 Intel(R) Xeon(R) 2.30 GHz CPU E5-2699 v3s processors and 64 GB RAM.

C.4.1.1 Small, well-tested programs

The IntroClass benchmark [149] consists of 998 small defective C programs (maximum 25 lines of code) with multiple test suites, intended for evaluating automatic program repair tools. Because the programs are small, it is computationally feasible to run SOSRepair on all defects multiple times, for experiments that require several rounds of execution on the whole benchmark. Since our main focus is applicability to real-world defects, we use the IntroClass benchmark for tuning experiments, and to compare with prior work that cannot scale to real-world defects.

Defects. The IntroClass benchmark consists of 998 defects from solutions submitted by undergraduate students to six small C programming assignments in an introductory C programming course. Each problem class (assignment) is associated with two independent test suites: One that is written by the instructor of the course (the black-box test suite), and one that is automatically generated by KLEE [33], a symbolic execution tool that automatically generates tests (the white-box test suite). Figure C.6 shows the number of defects in each program assignment group that fail at least one test case from the *black-box* test suite. The total number of such defects is 778.

Patch quality. For all repair experiments on IntroClass, we provide the black-box tests to the repair technique to guide the search for a patch. We then use the white-box test suite to measure patch quality, in terms of the percent of held-out tests the patched program passes (higher is better).

C.4.1.2 Large, real-world programs

The ManyBugs [149] benchmark consists of 185 defects taken from nine large, open-source C projects, commonly used to evaluate automatic program repair tools (e.g., [177, 190, 224, 282]).

Defects. The first four columns of Figure 3.2 show the project, size of source code, number of developer-written tests, and the number of defective versions of

program	kLOC	tests	defects	patched
gmp	145	146	2	0
gzip	491	12	4	0
libtiff	77	78	9	8
lighttpd	62	295	5	1
php	1,099	8,471	39	9
python	407	355	4	2
wireshark	2,814	63	2	2
total	5,095	9,420	65	22

Figure C.4: Subject programs and defects in our study, and the number of each for which SOSRepair generates a patch.

program	snippets	snippet size (# characters)		variables		# of functions called in the snippet		constraints per snippet		time to build the DB (h)
		mean	median	mean	median	mean	median	mean	median	
gmp	6,003	95.4	88	4.0	4	0.9	0	32.7	3	26.3
gzip	2,028	103.2	93	2.6	2	1.1	1	25.4	2	2.3
libtiff	3,010	114.8	108	3.0	3	1.2	1	29.9	2	5.8
lighttpd	797	90.6	82	2.0	2	1.4	1	24.8	2	2.3
php	22,423	113.5	100	2.7	2	1.4	1	19.8	2	51.6
python	20,960	116.1	108	2.4	2	1.0	1	26.9	1	41.9
wireshark	90,418	157.7	145	4.3	4	1.6	1	6.4	2	115.1

Figure C.5: The code snippet database SOSRepair generates for each of the ManyBugs programs. SOSRepair generated a total of 145,639 snippets, with means of 140 characters, 4 variables, 1 function call, and 13 SMT constraints. On average, SOSRepair builds the database in 35 hours, using a single thread.

the ManyBugs programs we use to evaluate SOSRepair. Prior work [195] argues for explicitly defining *defect classes* (the types of defects that can be fixed by a given repair method) while evaluating repair tools, to allow for fair comparison of tools on comparable classes. For instance, Angelix [190] cannot fix the defects that require adding a new statement or variable, and therefore all defects that require such modification are excluded from its defect class. For SOSRepair, we define a more general defect class that includes all the defects that can be fixed by editing one or more consecutive lines of code in one location, and are supported by BugZoo

(version 2.1.29) [270]. As mentioned in Section C.3.2, we use Docker containers managed by BugZoo to run experiments in a reproducible fashion. BugZoo supports ManyBugs scenarios that can be configured on a modern, 64-bit Linux system; we therefore exclude 18 defects from `valgrind` and `fbc`, which require the 32-bit Fedora 13 virtual machine image originally released with ManyBugs. Further, automatically fixing defects that require editing multiple files or multiple locations within a file is beyond SOSRepair's current capabilities. We therefore limit the scope of SOSRepair's applicability only to the defects that require developers to edit one or more consecutive lines of code in a single location. In theory, SOSRepair can be used to find multi-location patches, but considering multiple locations increases the search space and is beyond the scope of this paper.

SOSRepair's defect class includes 65 of the 185 ManyBugs defects. We use method-level fault localization by limiting SOSRepair's fault localization to the method edited by the developer's patch, which is sometimes hundreds of lines long. We construct a single snippet database (recall Section C.3) per project from the oldest version of the buggy code among all the considered defects. Therefore, the snippet database contains none of the developer-written patches.

Figure C.5 shows, for each ManyBugs program, the mean and median snippet size, the number of variables in code snippets, the number of functions called within the snippets, the number of constraints for the code snippets stored in the database, and the time spent on building the database. For each program, SOSRepair generates thousands of snippets, and for each snippet, on average, KLEE generates tens of SMT constraints. SOSRepair generated a total of 145,639 snippets, with means of 140 characters, 4 variables, 1 function call, and 13 SMT constraints. The database generation is SOSRepair's most time-consuming step, which only needs to happen once per project. The actual time to generate the database varies based on the size of the project. It takes from 2.3 hours for `gzip` up to 115 hours for `wireshark`, which is

the largest program in the ManyBugs benchmark. On average, it takes 8.2 seconds to generate each snippet. However, we collected these numbers using a single thread. This step is easily parallelizable, representing a significant performance opportunity in generating the database. We set the snippet granularity to 3–7 lines of code, following the results of our granularity experiments (Section C.4.3) and previous work on code redundancy [84].

Patch quality. A key concern in automated program repair research is the *quality* of the produced repairs [227, 252]. One mechanism for objectively evaluating patch quality is via independent test suites, held out from patch generation. The defects in ManyBugs are released with developer-produced test suites of varying quality, often with low coverage of modified methods. Therefore, we construct additional held-out test suites to evaluate the quality of generated patches. For a given defect, we automatically generate unit tests for all methods modified by either the project's developer or by at least one of the automated repair techniques in our evaluation. We do this by constructing small driver programs that invoke the modified methods:

- Methods implemented as part of an *extension* or *module* can be directly invoked from a driver's `main` function (e.g., the `substr_compare` method of `php string` module.)

- Methods implemented within *internal libraries* are invoked indirectly by using other functionality. For example, the method `do_inheritance_check_on_method` of `zend_compile` library in `php` is invoked by creating and executing `php` programs that implement inheritance. For such methods, the driver's `main` function sets the values of requisite global variables and then calls the functionality that invokes the desired method.

We automatically generate random test inputs for the driver programs that then invoke modified methods. We generate inputs until either the tests fully cover the target

method or until adding new test inputs no longer significantly increases statement coverage. For four `php` and two `lighttpd` scenarios for which randomly generated test inputs were unable to achieve high coverage, we manually added new tests to that effect. For `libtiff` methods requiring tiff images as input, we use 7,214 tiff images randomly generated and released by the AFL fuzz tester [4]. We use the developer-patched behavior to construct test oracles, recording logged, printed, and returned values and exit codes as ground truth behavior. If the developer-patched program crashes on an input, we treat the crash as the expected behavior.

We release these generated test suites (along with all source code, data, and experimental results) to support future evaluations of automated repair quality on ManyBugs. All materials may be downloaded from `https://github.com/squaresLab/SOSRepair-Replication-Package`. This release is the first set of independently-generated quality-evaluation test suites for ManyBugs.

Baseline approaches. We compare to three previous repair techniques that have been evaluated on (subsets) of ManyBugs, relying on their public data releases. Angelix [190] is a state-of-the-art semantic program repair approach; Prophet [177] is a more recent heuristic technique that instantiates templated repairs [175], informed by machine learning; and GenProg [150] uses genetic programming to combine statement-level program changes in a repair search. GenProg has been evaluated on all 185 ManyBugs defects; Angelix, on 82 of the 185 defects; Prophet, on 105 of 185. Of the 65 defects that satisfy SOSRepair's defect class, GenProg is evaluated on all 65 defects, Angelix on 30 defects, and Prophet on 39 defects.

C.4.2 Comparison to SearchRepair

To substantiate SOSRepair's improvement over previous work in semantic search-based repair, we empirically compare SOSRepair's performance with SearchRepair [122]. Because SearchRepair does not scale to the ManyBugs programs, we conduct this

experiment on the IntroClass dataset (Section C.4.1.1). We use the black-box tests to guide the search for repair, and the white-box tests to evaluate the quality of the produced repair.

Figure C.6 shows the number of defects patched by each technique. SOSRepair patches more than twice as many defects as SearchRepair (346 versus 150, out of the 778 total repairs attempted). This difference is statistically significant based on Fisher's exact test ($p < 10^{-15}$). The bottom row shows the mean percent of the associated held-out test suite passed by each patched program. Note that SOSRepair's average patch quality is slightly lower than SearchRepair's (91.5% versus 97.3%). However, 239 of the 346 total SOSRepair patches pass 100% of the held-out tests, constituting substantially more very high-quality patches than SearchRepair finds total (150). Overall, however, semantic search-based patch quality is quite high, especially as compared to patches produced by prior techniques as evaluated in the prior work: AE [282] finds patches for 159 defects with average quality of 64.2%, TrpAutoRepair [224] finds 247 patches with 72.1% quality, and GenProg [284] finds 287 patches with average quality of 68.7% [122]. Overall, SOSRepair outperforms these prior techniques in expressive power (number of defects repaired, at 346 of 778), and those patches are of measurably higher quality.

C.4.3 Snippet Granularity

Snippet granularity informs the size and preparation of the candidate snippet database, as well as SOSRepair's expressiveness. Low granularity snippets may produce prohibitively large databases and influence patch quality. High granularity (i.e., larger) snippets lower the available redundancy (previous work suggests that the highest code redundancy is found in snippets of 1–7 lines of code [84]) and may reduce the probability of finding fixes. Both for tuning purposes and to assess one of our underlying hypotheses, we evaluate the effect of granularity on repair success

problem class	defects	SearchRepair	SOSRepair
checksum	29	0	3
digits	91	0	24
grade	226	2	37
median	168	68	87
smallest	155	73	120
syllables	109	4	75
total	778	150	346
mean quality		97.3%	91.5%

Figure C.6: Number of defects repaired by SearchRepair and SOSRepair in IntroClass dataset. "Mean quality" denotes the mean percent of the associated held-out test suite passed by each patched programs.

and patch quality by systematically altering the granularity level of both the code snippets in the SOSRepair database and the buggy snippet to be repaired. Because this requires a large number of runs on many defects to support statistically significant results, and to reduce the confounds introduced by real-world programs, we conduct this experiment on the IntroClass dataset, and use SOSRepair to try to repair all defects in the dataset using granularity level configuration of 1–3 lines, 3–7 lines, and 6–9 lines of code.

Figure C.7 shows the number of produced patches, the number of those patches that pass *all* the held-out tests, and the mean percent of held-out test cases that the patches pass, by granularity of the snippets in the SOSRepair database. The granularity of 3–7 lines of code produces the most patches (346 versus 188 and 211 with other granularities), and the most patches that pass all the held-out tests (239 versus 120 and 125 with other granularities). Fisher's exact test confirms that these differences are statistically significant (all $p < 10^{-70}$).

While the number of patches that pass all defects is significantly higher for the 3–7 granularity, and the fraction of patches that pass all held-out tests is higher for that

program	patches			patches passing all held-out tests			mean % of held-out tests passing		
	1–3	3–7	6–9	1–3	3–7	6–9	1–3	3–7	6–9
checksum	0	3	8	0	3	8	—	100.0%	100.0%
digits	26	24	17	14	9	5	91.5%	89.5%	92.9%
grade	1	37	2	1	37	2	100.0%	100.0%	100.0%
median	14	87	52	1	63	44	84.5%	95.0%	95.5%
smallest	60	120	132	27	57	54	80.4%	82.2%	78.5%
syllables	87	75	17	77	70	12	97.0%	98.6%	97.0%
Total	188	346	211	120	239	125			

Figure C.7: A comparison of applying SOSRepair to IntroClass defects with three different levels of granularity: 1–3, 3–7, and 6–9 lines of code.

granularity (69.1% for 3–7, 63.8% for 1–3, and 59.2% for 6–9), the mean patch quality is similar for all the three levels of granularity. We hypothesize that this observation may be a side-effect of the small size of the programs in the IntroClass benchmark and the high redundancy induced by many defective programs in that benchmark attempting to satisfy the same specification. We suspect this observation will not extend to benchmarks with more diversity and program complexity, and thus make no claims about the effect of granularity on average quality.

We configure our database in subsequent experiments to use snippets of 3–7 lines, as these results suggest that doing so may provide a benefit in terms of expressive power. The results of this study may not immediately extend to large, real-world programs; we leave further studies exploring repair granularity for large programs to future work.

C.4.4 Repair of large, real-world programs

A key contribution of our work is a technique for semantic search-based repair that scales to real-world programs; we therefore evaluate SOSRepair on defects from ManyBugs that fall into its defect class (as described in Section C.4.1.2). The

"patched" column in Figure C.4 summarizes SOSRepair's ability to generate patches. Figure 3.5 presents repair effectiveness and quality for all considered defects in the class, comparing them with patches produced by previous evaluations of Angelix, Prophet, and GenProg. Figure 3.5 enumerates defects for readability and maps each "program ID" to a revision pair of the defect and developer-written repair.

C.4.4.1 Repair expressiveness and applicability

SOSRepair patches 22 of the 65 defects that involved modifying consecutive lines by the developer to fix those defects. The Angelix, Prophet, and GenProg columns in Figure 3.5 indicate which approaches succeed on patching those defects (✘ for not patched, and NA for not attempted, corresponding to defects outside the defined defect class for a technique). There are 5 defects that all four techniques patch. SOSRepair is the only technique that repaired `libtiff-4`. SOSRepair produces patches for 3 defects that Angelix cannot patch, 5 defects that Prophet cannot patch, and 6 defects that GenProg cannot patch. These observations corroborate results from prior work on small programs, which showed that semantic search-based repair could target and repair defects that other techniques cannot [122].

Even though efficiency is not a focus of SOSRepair's design, we measured the amount of time required to generate a patch with SOSRepair. On average, it took SOSRepair 5.25 hours to generate patches reported in Figure 3.5. Efficiency is separate from, and secondary to the ability to produce patches and can be improved by taking advantage of parallelism and multithreading in SOSRepair's implementation. On average, 57.6% of the snippets in the database (satisfying type constraints) matched the SMT query described in Section C.2.4. Of the repaired defects, seven involve insertion, seven involve replacement, and eight involve deletion.

C.4.4.2 Repair effectiveness and quality

Figure 3.5 shows the percent of evaluation tests passed by the SOSRepair, Angelix, Prophet, and GenProg patches. "Coverage" is the average statement-level coverage of the generated tests on the methods modified by either the developer or by at least one automated repair technique in our evaluation. SOSRepair produces more patches (9, 41%) that pass all independent tests than Angelix (4), Prophet (5) and, GenProg (4). For the defects patched in-common by SOSRepair and other techniques, Angelix and SOSRepair patch 9 of the same defects; both SOSRepair and Angelix produce 4 patches that pass all evaluation tests on this set. Prophet and SOSRepair patch 11 of the same defects; both SOSRepair and Prophet produce 5 patches that pass all evaluation tests on this set. GenProg and SOSRepair patch 16 of the same defects; 4 out of these 16 GenProg patches and 8 SOSRepair patches pass all evaluation tests. Thus, SOSRepair produces more patches that pass all independent tests than GenProg, and as many such patches as Angelix and Prophet. This suggests that semantic code search is a promising approach to generate high-quality repairs for real defects, and that it has potential to repair defects that are outside the scope of other, complementary repair techniques.

C.4.4.3 Improving patch quality through fault localization

Although these baseline results are promising, most of the patches previous semantic search-based repair produced on small program defects passed all held-out tests [122]. We investigated why SOSRepair patch quality is lower than this high bar. We hypothesized that two possible reasons are that real-world buggy programs do not contain code that can express the needed patch, or that fault localization imprecision hampers SOSRepair success. Encouragingly, anecdotally, we found that many buggy programs do contain code that can express the developer patch. However, fault localization is the more likely culprit. For example, for `gmp-1`, fault localization reports

59 lines as equally-highly suspicious, including the line modified by the developer, but as part of its breadth-first strategy, SOSRepair only tries 10 of these 59.

We further observed that in some cases, more than one mapping between variables satisfies the query, but only one results in a successful patch. Since trying all possible mappings is not scalable, SOSRepair only tries the first mapping selected by the solver. Including more variables in the mapping query increases the number of patch possibilities, but also the complexity of the query.

We created SOSRepair$^{\oplus}$, a semi-automated version of SOSRepair that can take hints from the developer regarding fault location and variables of interest. SOSRepair$^{\oplus}$ differs from SOSRepair in the following two ways:

1. SOSRepair uses spectrum-based fault localization [117] to identify candidate buggy code regions. SOSRepair$^{\oplus}$ uses a manually-specified candidate buggy code region. In our experiments, SOSRepair$^{\oplus}$ uses the location of the code the developer modified to patch the defect as its candidate buggy code region, simulating the developer suggesting where the repair technique should try to repair a defect.

2. SOSRepair considers all live variables after the insertion line in its query. While multiple mappings may exist that satisfy the constraints, not all such mappings may pass all the tests. SOSRepair uses the one mapping the SMT solver returns. SOSRepair$^{\oplus}$ can be told which variables not to consider, simulating the developer suggesting to the repair technique which variables likely matter for a particular defect. A smaller set of variables of interest increases the chance that the mapping the SMT solver returns and SOSRepair$^{\oplus}$ tries is a correct one. We found that for 6 defects (`gzip-1`, `libtiff-4`, `libtiff-8`, `php-10`, `php-12`, and `gmp-1`), SOSRepair failed to produce a patch because it attempted an incorrect mapping. For these 6 defects, we instructed SOSRepair$^{\oplus}$ to reduce the variables of interest to just those variables used in the developer-written patch.

On our benchmark, SOSRepair$^{\oplus}$ patches 23 defects and 16 (70%) of them pass all independent tests. While it is unsound to compare SOSRepair$^{\oplus}$ to prior, fully-automated techniques, our conclusions are drawn only from the comparison to SOSRepair; the quality results for the SOSRepair$^{\oplus}$-patched defects for the prior tools in Figure 3.5 are only for reference.

Our experiments show that precise fault localization allows SOSRepair$^{\oplus}$ to patch 7 additional defects SOSRepair could not (bottom of Figure 3.5), and to improve the quality of 3 of SOSRepair's patches. Overall, 9 new patches pass 100% of the independent tests.

SOSRepair and SOSRepair$^{\oplus}$ sometimes attempt to patch defects at different locations: SOSRepair using spectrum-based fault localization and SOSRepair$^{\oplus}$ at the location where the developer patched the defect. For 6 defects, SOSRepair finds a patch, but SOSRepair$^{\oplus}$ does not. Note that defects can often be patched at multiple locations, and developers do not always agree on a single location to patch a particular defect [23]. Thus, the localization hint SOSRepair$^{\oplus}$ receives is a heuristic, and may be neither unique nor optimal. In each of these 6 cases, the patch SOSRepair finds it at an alternate location than where the developer patched the defect.

Because SOSRepair and SOSRepair$^{\oplus}$ sometimes patch at different locations, the patches they produce sometimes differ, and accordingly, so does the quality of those patches. In our experiments, in all but one case (`php-5`) SOSRepair$^{\oplus}$ patches were at least as high, or higher quality than SOSRepair patches for the same defect.

We conclude that research advancements that produce more accurate FL or elicit guidance from developers in a lightweight manner are likely to dramatically improve SOSRepair performance. Additionally, input (or heuristics) on which variables are likely related to the buggy functionality (and are thus appropriate to consider) could limit the search to a smaller but more expressive domain, further improving SOSRepair.

C.4.5 Example patches

In this section, we present several SOSRepair patches produced on the ManyBugs defects (Section C.4.4), comparing them to developer patches and those produced by other tools. Our goal is not to be comprehensive, but rather to present patches highlighting various design decisions.

Example 1: python-1. The `python` interpreter at revision #69223 fails a test case concerning a variable that should never be negative. The developer patch is as follows:

```
}
+ if (timeout < 0) {
+     PyErr_SetString(PyExc_ValueError,
+           "timeout must be non-negative");
+     return NULL;
+ }
seconds = (long)timeout;
```

Fault localization correctly identifies the developer's insertion point for repair. Several snippets in the `python` project perform similar functionality to the fix, including the following, from the `IO` module:

```
if (n < 0) {
PyErr_SetString(PyExc_ValueError,
"invalid key number");
return NULL;
}
```

SOSRepair correctly maps variable `n` to `timeout` and inserts the code to repair the defect. Although the error message is not identical, the functionality is, and suitable to satisfy the developer tests. However, unlike the developer tests, the generated tests do consider the error message, explaining the patch's relatively low success on the held-out tests. Synthesizing good error messages is an open problem; such a semantically meaningful patch could still assist developers in more quickly addressing the underlying defect [281].

GenProg did not patch this defect; Angelix was not attempted on it, as the defect is outside its defect class. The Prophet patch modifies an if-check elsewhere in the code to include a tautological condition:

```
- if ((!rv)) {
+ if((!rv) && !(1)) {
if (set_add_entry((PySetObject *)...
```

This demonstrates how techniques that do not delete directly can still do so, motivating our explicit inclusion of deletion.

Example 2: php-2. We demonstrate the utility of explicit deletion with our second example, from `php-2` (recall Figure 3.5). At the buggy revision, `php` fails two test cases because of an incorrect value modification in its `string` module. Both the developer and SOSRepair delete the undesired functionality:

```
-  if (len > s1_len - offset) {
-    len = s1_len - offset;
-  }
```

Angelix and Prophet correctly eliminate the same functionality by modifying the `if` condition such that it always evaluates to `false`. GenProg inserts a `return;` statement in a different method.

Example 3: php-1. Finally, we show a SOSRepair patch that captures a desired semantic effect while syntactically different from the human-written repair. Revision 74343ca506 of `php-1` (recall Figure 3.5) fails 3 test cases due to an incorrect condition around a loop `break`, which the developer modifies:

```
- if (just_read < toread) {
+ if (just_read == 0) {
break;
}
```

This defect inspired our illustrative example (Section C.2.1). Using default settings, SOSRepair first finds a patch identical to the developer fix. To illustrate, we present a different but similar fix that SOSRepair finds if run beyond the first repair:

```
if ((int)box_length <= 0) {
break;
}
```

SOSRepair maps `box_length` to `just_read`, and replaces the buggy code. In this code, `just_read` is only ever greater than or equal to zero, such that this patch is acceptable.

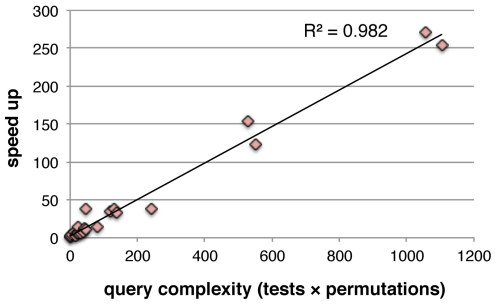


Figure C.8: The speedup of the new encoding approach over the previous approach grows with query complexity.

Angelix and Prophet were not attempted on this defect; GenProg deletes other functionality.

C.4.6 Query encoding performance

To answer our final two research questions, we isolate and evaluate two key novel features of SOSRepair. First, this section evaluates the performance improvements gained by SOSRepair's novel query encoding approach. Second, Section C.4.7 evaluates the effects of SOSRepair's negative profile refinement approach on reducing the search space.

In the repair search problem, query complexity is a function of the number of test inputs through a region and the number of possible mappings between a buggy region and the repair context. To understand the differences between SOSRepair's and the old approach's encodings, consider a buggy snippet $\mathcal{C}$ with two input variables a and b and a single output variable c. Suppose $\mathcal{C}$ is executed by two tests, t_1 and t_2. And suppose $\mathcal{S}$ is a candidate repair snippet with two input variables x and y, a single output variable z, and path constraints φ_c generated by the symbolic execution engine. SOSRepair's encoding uses *location variables* to discover a valid mapping between variables a, b and x, y that satisfy φ_c constraints for both test cases t_1 and t_2, with a single query (recall Section C.2.4.1). Meanwhile, the prior approach [122]

traverses all possible mappings between variables ($m_1 : (a = x) \wedge (b = y) \wedge (c = z)$ and $m_2 : (a = y) \wedge (b = x) \wedge (c = z)$), and creates a query for every test case, for every possible variable mapping. A satisfiable query implies its mapping is valid for that particular test. For example, to show that mapping m_1 is a valid mapping, two queries are required (one for t_1 and one for t_2), and only if both are satisfiable is m_1 considered valid. The number of queries required for this approach grows exponentially in the number of variables, as there is an exponential number of mappings (permutation) of the variables. In our example, there are two possible mappings and two tests, so four queries are required, unlike SOSRepair's one.

To evaluate the performance impact of SOSRepair's new encoding, we reimplement the previous encoding approach [122]. We then compare SMT solver speed on the same repair questions using each encoding. Running on two randomly-selected ManyBugs defects, we measured the response time of the solver on more than 10,000 queries for both versions of encoding techniques. Figure C.8 shows the speed up using the new encoding as compared to the old encoding, as a function of query complexity (number of tests times the number of variable permutations). The new encoding approach delivers a significant speed up over the previous approach, and the speed up increases linearly with query complexity ($R^2 = 0.982$).

Looking at the two approaches individually, query time increases linearly with query complexity (growing slowly slope-wise, but with a very high $R^2 = 0.993$) with the previous encoding, and is significantly more variable with the new encoding and does not appear linearly related to query complexity ($R^2 = 0.008$). Overall, Figure C.8 shows the speed up achieved with the new encoding, and its linear increase as query complexity grows.

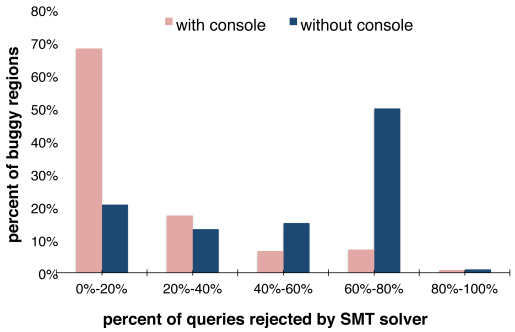


Figure C.9: Fraction of defects that can reject fractions of the search space (measured via SMT queries) using only iteratively-constructed negative examples. Profile refinement improves scalability by reducing the number of candidate snippets to consider. Console output that relies on symbolic values affects this performance.

C.4.7 Profile refinement performance

The profile refinement approach (recall Section C.2.5) uses negative tests to iteratively improve a query, reduce the number of attempted candidate snippets, and repair defects without covering passing test cases. By default, SOSRepair uses the automated, iterative query refinement on all defects whenever at least one faulty region under consideration is covered only by negative test cases. In our experiments, for 2 ManyBugs defects (`libtiff-8` and `lighttpd-2`), the patches SOSRepair and SOSRepair$^{\oplus}$ produce cover a region only covered by negative test cases, though SOSRepair and SOSRepair$^{\oplus}$ use the refinement process while attempting to patch other defects as well.

In this experiment, we evaluate the effect of iterative profile refinement using negative examples on the size of the considered SMT search space. We conduct this experiment on a subset of the IntroClass dataset to control for the effect of symbolic execution performance (which is highly variable on the real-world programs in ManyBugs). We ran SOSRepair on all the defects in the `median`, `smallest`, and `grade` programs, only using the initially failing test cases, with profile refinement, for repair. For every buggy line selected by the fault localization and expanded into a region

with granularity of 3–7 lines of code, we measured the number of candidate snippets in the database that can be rejected by the SMT-solver (meaning the patch need not be dynamically tested to be rejected, saving time) using only negative queries.

Figure C.9 shows the percent of the search space excluded after multiple iterations for all buggy regions. For example, the first bar shows that on 68% of buggy regions tried, fewer than 20% of candidate snippets were eliminated by the solver when only negative tests are available, leaving more than 80% of possible candidates for dynamic evaluation. We find that approach effectiveness depends on the nature of the defect and snippets. In particular, the approach performs poorly when desired snippet behavior involves console output that depends on a symbolic variable. This makes sense: KLEE produces random output in the face of symbolic console output, and such output is uninformative in specifying undesired behavior. Our results show that on 14% of the defects (that are dependent on console output), more than 40% of database snippets can be rejected using only the test cases that the program initially failed. We also transformed the defects in the dataset to capture console output by variable assignments, treating those variables as the output (rather than the console printout); Figure C.9 also shows the results of running the same study on the modified programs. More than 40% of the possible snippets can be eliminated for 66% of the preprocessed programs. Overall, profile refinement can importantly eliminate large amounts of the search space, but its success depends on the characteristics of the code under repair.

C.4.8 Threats and limitations

Even though SOSRepair works on defects that require developers to modify a single (potentially multi-line) location in the source code, we ensure that it generalizes to all kinds of defects belonging to large unrelated projects by evaluating SOSRepair on a subset of the ManyBugs benchmark [149], which consists of real-world, real-developer

defects, and is used extensively by prior program repair evaluations [149, 177, 190, 199, 224, 282]. The defects in our evaluation also cover the novel aspects of our approach, e.g., defects with only negative profiles, console output, and various edit types.

Our work inherits KLEE's limitations: SOSRepair cannot identify snippets that KLEE cannot symbolically execute, impacting patch expressiveness nevertheless, the modified buggy code can include KLEE-unsupported constructs, such as function pointers. Note that this limitation of KLEE is orthogonal to our repair approach. As KLEE improves in its handling of more complex code, so will SOSRepair. Our discussion of other factors influencing SOSRepair success (recall Section C.4.4) suggests directions for improving applicability and quality.

Our experiments limit the database of code snippets to those found in the same project, based on observations of high within-project redundancy [14]. Anecdotally, we have observed SOSRepair failing to produce a patch when using snippets only from the same project, but succeeding with a correct patch when using snippets from other projects. For example, for `gzip-1` defect, the code in `gzip` lacks the necessary snippet to produce a patch, but that snippet appears in the `python` code. Extending SOSRepair to use snippets from other projects could potentially improve SOSRepair's effectiveness, but also creates new scalability challenges, including handling code snippets that include custom-defined, project-specific types and structures.

Precisely assessing patch quality is an unsolved problem. As with other repair techniques guided by tests, we use tests, a partial specification, to evaluate the quality of SOSRepair's patches. Held-out, independently generated or written test suites represent the state-of-the-art of patch quality evaluation [252], along with manual inspection [177, 227]. Although developer patches (which we use as a functional oracle) may contain bugs, in the absence of a better specification, evaluations such as ours must rely on the developers.

We conduct several experiments (e.g., Sections C.4.3 and C.4.7) on small programs from the IntroClass benchmark [149], since these experiments require controlled, large-scale executions of SOSRepair. Even though these experiments provide valuable insights, their results may not immediately extend to large, real-world programs.

To mitigate the risk of errors in our implementation or setup, we publicly release our code, results, and new test suites to support future evaluation, reproduction, and extension. All materials are available at:
`https://github.com/squaresLab/SOSRepair` (SOSRepair's implementation), and
`https://github.com/squaresLab/SOSRepair-Replication-Package`
(SOSRepair's replication package).

C.5 Contributions

Automated program repair may reduce software production costs and improve software quality, but only if it produces high-quality patches. While semantic code search can produce high-quality patches [122], such an approach has never been demonstrated on real-world programs. In this study, we have designed SOSRepair, a novel approach to using semantic code search to repair programs, focusing on extending expressiveness to that of real-world C programs and improving the search mechanism's scalability. We evaluate SOSRepair on 65 defects in large, real-world C programs, such as `php` and `python`. SOSRepair produces patches for 22 (34%) of the defects, and 9 (41%) of those patches pass 100% of independently-generated, held-out tests. SOSRepair repairs a defect no prior techniques have, and produces higher-quality patches. In a semi-automated approach that manually specifies the fault's location, SOSRepair patches 23 defects, of which 16 (70%) pass all independent tests. Our results suggest semantic code search is a promising approach for automatically repairing real-world defects.

BIBLIOGRAPHY

[1] Abd-El-Malek, Michael, Ganger, Gregory R., Goodson, Garth R., Reiter, Michael K., and Wylie, Jay J. Fault-scalable Byzantine fault-tolerant services. In *ACM Symposium on Operating Systems Principles (SOSP)* (Brighton, UK, 2005), pp. 59–74.

[2] Abreu, Rui, Zoeteweij, Peter, and Gemund, Arjan JC Van. On the accuracy of spectrum-based fault localization. In *IEEE Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION)* (Windsor, UK, Sept. 2007), pp. 89–98.

[3] Ackling, Thomas, Alexander, Bradley, and Grunert, Ian. Evolving patches for software repair. In *Annual Conference on Genetic and Evolutionary Computation (GECCO)* (Dublin, Ireland, 2011), pp. 1427–1434.

[4] American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2018.

[5] Afzal, Afsoon, Motwani, Manish, Stolee, Kathryn, Brun, Yuriy, and Le Goues, Claire. SOSRepair: Expressive] Semantic Search for Real-World Program Repair. *IEEE Transactions on Software Engineering* (2021).

[6] Agrawal, Hiralal, Horgan, Joseph R., London, Saul, and Wong, W. Eric. Fault localization using execution slices and dataflow tests. In *Intl. Symposium on Software Reliability Engineering (ISSRE)* (Toulouse, France, 1995), pp. 143–151.

[7] Alkhalaf, Muath, Aydin, Abdulbaki, and Bultan, Tevfik. Semantic differential repair for input validation and sanitization. In *Intl. Symposium on Software Testing and Analysis (ISSTA)* (San Jose, CA, USA, July 2014), pp. 225–236.

[8] Ammann, Paul, and Offutt, Jeff. *Introduction to Software Testing*, 1 ed. Cambridge University Press, New York, NY, USA, 2008.

[9] Angell, Rico, Johnson, Brittany, Brun, Yuriy, and Meliou, Alexandra. Themis: Automatically testing software for discrimination. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) Demo track* (Nov. 2018), pp. 871–875.

[10] Arcuri, Andrea, and Briand, Lionel. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Honolulu, HI, USA, 2011), pp. 1–10.

[11] Assiri, Fatmah Yousef, and Bieman, James M. Fault localization for automated program repair: Effectiveness, performance, repair correctness. *Software Quality Journal 25*, 1 (2017), 171–199.

[12] Association, European Computer Manufacturer's. ECMA standards. `https://ecma-international.org/publications/standards/Standard.htm`, 2018.

[13] Atlassian. Clover code coverage tool. `https://www.atlassian.com/software/clover`, 2016.

[14] Barr, Earl T., Brun, Yuriy, Devanbu, Premkumar, Harman, Mark, and Sarro, Federica. The plastic surgery hypothesis. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (November 2014), pp. 306–317.

[15] Beschastnikh, Ivan, Brun, Yuriy, Abrahamson, Jenny, Ernst, Michael D., and Krishnamurthy, Arvind. Unifying FSM-inference algorithms through declarative specification. In *5th International Conference on Software Engineering (ICSE)* (San Francisco, CA, USA, May 2013), pp. 252–261.

[16] Beschastnikh, Ivan, Brun, Yuriy, Abrahamson, Jenny, Ernst, Michael D., and Krishnamurthy, Arvind. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering (TSE) 41*, 4 (April 2015), 408–428.

[17] Beschastnikh, Ivan, Brun, Yuriy, Ernst, Michael D., and Krishnamurthy, Arvind. Inferring Models of Concurrent Systems from Logs of their Behavior with CSight. In *International Conference on Software Engineering (ICSE)* (Hyderabad, India, June 2014), pp. 468–479.

[18] Beschastnikh, Ivan, Brun, Yuriy, Schneider, Sigurd, Sloan, Michael, and Ernst, Michael D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Szeged, Hungary, September 2011), pp. 267–277.

[19] Bhatia, Sahil, Kohli, Pushmeet, and Singh, Rishabh. Neuro-symbolic program corrector for introductory programming assignments. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2018), pp. 60–70.

[20] Bhavsar, Hetal, and Ganatra, Amit. A comparative study of training algorithms for supervised machine learning. *International Journal of Soft Computing and Engineering (IJSCE) 2*, 4 (2012), 2231–2307.

[21] Bird, Christian, Bachmann, Adrian, Aune, Eirik, Duffy, John, Bernstein, Abraham, Filkov, Vladimir, and Devanbu, Premkumar. Fair and balanced?: Bias in bug-fix datasets. In *European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Amsterdam, The Netherlands, August 2009), pp. 121–130.

[22] Blasi, Arianna, Goffi, Alberto, Kuznetsov, Konstantin, Gorla, Alessandra, Ernst, Michael D., Pezzè, Mauro, and Castellanos, Sergio Delgado. Translating code comments to procedure specifications. In *International Symposium on Software Testing and Analysis (ISSTA)* (Amsterdam, Netherlands, 2018), pp. 242–253.

[23] Böhme, Marcel, Soremekun, Ezekiel Olamide, Chattopadhyay, Sudipta, Ugherughe, Emamurho, and Zeller, Andreas. Where is the bug and how is it fixed? An experiment with practitioners. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Sept. 2017), pp. 117–128.

[24] Boyapati, Chandrasekhar, Khurshid, Sarfraz, and Marinov, Darko. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)* (Rome, Italy, 2002), pp. 123–133.

[25] Brandenburg, Franz J, Gleißner, Andreas, and Hofmeier, Andreas. Comparing and aggregating partial orders with kendall tau distances. *Discrete Mathematics, Algorithms and Applications 5*, 02 (2013), 1360003.

[26] Brandenburg, Franz J, Gleißner, Andreas, and Hofmeier, Andreas. The nearest neighbor spearman footrule distance for bucket, interval, and partial orders. *Jour. of Combinatorial Optimization 26*, 2 (2013), 310–332.

[27] Briand, Lionel C, Labiche, Yvan, and Liu, Xuetao. Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)* (2007), IEEE, pp. 137–146.

[28] Brubaker, Chad, Jana, Suman, Ray, Baishakhi, Khurshid, Sarfraz, and Shmatikov, Vitaly. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy (S&P)* (2014), pp. 114–129.

[29] Brun, Yuriy, Barr, Earl, Xiao, Ming, Le Goues, Claire, and Devanbu, Prem. Evolution vs. intelligent design in program patching. Tech. Rep. `https://escholarship.org/uc/item/3z8926ks`, UC Davis: College of Engineering, 2013.

[30] Brun, Yuriy, and Meliou, Alexandra. Software fairness. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) New Ideas and Emerging Results track* (Lake Buena Vista, FL, USA, Nov. 2018), pp. 754–759.

[31] Bryant, Antony, and Charmaz, Kathy. *The SAGE handbook of grounded theory.* SAGE Publications Ltd, 2007.

[32] Burges, Chris, Shaked, Tal, Renshaw, Erin, Lazier, Ari, Deeds, Matt, Hamilton, Nicole, and Hullender, Greg. Learning to rank using gradient descent. In *ACM International Conference on Machine Learning* (2005), pp. 89–96.

[33] Cadar, Cristian, Dunbar, Daniel, and Engler, Dawson. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, USA, 2008), pp. 209–224.

[34] Campos, José, Riboira, André, Perez, Alexandre, and Abreu, Rui. Gzoltar: An Eclipse plug-in for testing and debugging. In *IEEE/ACM International Conference on Automated Software Engineering* (2012), pp. 378–381.

[35] Carzaniga, Antonio, Gorla, Alessandra, Mattavelli, Andrea, Perino, Nicolò, and Pezzè, Mauro. Automatic recovery from runtime failures. In *ACM/IEEE International Conference on Software Engineering* (2013), pp. 782–791.

[36] Carzaniga, Antonio, Gorla, Alessandra, Perino, Nicolò, and Pezzè, Mauro. Automatic workarounds for web applications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2010), pp. 237–246.

[37] Charmaz, Kathy. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. SAGE Publications Ltd, 2006.

[38] Chen, Jie, Xu, Xiwei, Osterweil, Leon J., Zhu, Liming, Brun, Yuriy, Bass, Len, Xiao, Junchao, Li, Mingshu, and Wang, Qing. Using Simulation to Evaluate Error Detection Strategies: A Case Study of Cloud-Based Deployment Processes. *Journal of Systems and Software 110* (December 2015), 205–221.

[39] Chen, Liushan, Pei, Yu, and Furia, Carlo A. Contract-based program repair without the contracts. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Urbana, IL, USA, Nov. 2017), pp. 637–647.

[40] Chen, Zimin, Kommrusch, Steve James, Tufano, Michele, Pouchet, Louis-Noël, Poshyvanyk, Denys, and Monperrus, Martin. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019). [Online; accessed 20-March-2022].

[41] Chen, Zimin, Kommrusch, Steve James, Tufano, Michele, Pouchet, Louis-Noël, Poshyvanyk, Denys, and Monperrus, Martin. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE TSE* (2019).

[42] Cheng, Xi, Zhou, Min, Song, Xiaoyu, Gu, Ming, and Sun, Jiaguang. IntPTI: Automatic integer error repair with proper-type inference. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), pp. 996–1001.

[43] Christou, Steven. Cobertura code coverage tool. `https://cobertura.github.io/cobertura/`, 2015.

[44] Clang. A C language family frontend for LLVM. `https://clang.llvm.org/`.

[45] Cochran, Robert, D'Antoni, Loris, Livshits, Benjamin, Molnar, David, and Veanes, Margus. Program boosting: Program synthesis via crowd-sourcing. In *Symposium on Principles of Programming Languages (POPL)* (Mumbai, India, January 2015), pp. 677–688.

[46] Coker, Zack, and Hafiz, Munawar. Program transformations to fix C integers. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (San Francisco, CA, USA, 2013), pp. 792–801.

[47] Coles, Henry, Laurent, Thomas, Henard, Christopher, Papadakis, Mike, and Ventresque, Anthony. PIT: A practical mutation testing tool for java (demo). In *International Symposium on Software Testing and Analysis (ISSTA)* (Saarbrücken, Germany, 2016), ACM, pp. 449–452.

[48] Costea, Andreea, Zhu, Amy, Polikarpova, Nadia, and Sergey, Ilya. Concise read-only specifications for better synthesis of programs with pointers. In *ESOP* (2020), pp. 141–168.

[49] Cristian, Flaviu. Exception handling. Tech. Rep. RJ5724, IBM Research, 1987.

[50] Dallmeier, Valentin, Knopp, Nikolai, Mallon, Christoph, Hack, Sebastian, and Zeller, Andreas. Generating test cases for specification mining. In *International Symposium on Software Testing and Analysis (ISSTA)* (2010), pp. 85–96.

[51] Dallmeier, Valentin, Lindig, Christian, and Zeller, Andreas. Lightweight defect localization for Java. In *European Conference on Object Oriented Programming (ECOOP)* (Glasgow, UK, 2005), pp. 528–550.

[52] Dallmeier, Valentin, Zeller, Andreas, and Meyer, Bertrand. Generating fixes from object behavior anomalies. In *IEEE/ACM International Conference on Automated Software Engineering (ASE) short paper track* (Auckland, New Zealand, Nov. 2009), pp. 550–554.

[53] D'Antoni, Loris, Samanta, Roopsha, and Singh, Rishabh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification (CAV)* (Toronto, ON, Canada, July 2016), pp. 383–401.

[54] D'Antoni, Loris, Singh, Rishabh, and Vaughn, Michael. NoFAQ: Synthesizing command repairs from examples. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, Sept. 2017), pp. 582–592.

[55] De Moura, Leonardo, and Bjørner, Nikolaj. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[56] de Souza, Eduardo Faria, Le Goues, Claire, and Camilo-Junior, Celso Gonçalves. A novel fitness function for automated program repair based on source code checkpoints. In *Proceedings of the Genetic and Evolutionary Computation Conference* (New York, NY, USA, 2018), GECCO '18, p. 1443–1450.

[57] Deb, Kalyanmoy, Pratap, Amrit, Agarwal, Sameer, and Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation (TEVC) 6*, 2 (Apr. 2002), 182–197.

[58] Debroy, Vidroha, and Wong, W Eric. A consensus-based strategy to improve the quality of fault localization. *Software: Practice and Experience 43*, 8 (2013), 989–1011.

[59] DeMarco, Favio, Xuan, Jifeng, Berre, Daniel Le, and Monperrus, Martin. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA* (Hyderabad, India, 2014), pp. 30–39.

[60] DeMillo, Richard A, Lipton, Richard J, and Sayward, Frederick G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (1978), 34–41.

[61] Demsky, Brian, and Rinard, Martin. Automatic detection and repair of errors in data structures. *Acm sigplan notices 38*, 11 (2003), 78–95.

[62] Demsky, Brian, and Rinard, Martin. Data structure repair using goal-directed reasoning. In *International Conference on Software Engineering(ICSE)* (2005), pp. 176–185.

[63] Deng, Ke, Han, Simeng, Li, Kate J, and Liu, Jun S. Bayesian aggregation of order-based rank data. *Journal of the American Statistical Association 109*, 507 (2014), 1023–1039.

[64] Ding, Zhen Yu, Lyu, Yiwei, Timperley, Christopher, and Le Goues, Claire. Leveraging program invariants to promote population diversity in search-based automatic program repair. In *International Workshop on Genetic Improvement (GI)* (Montreal, QC, Canada, 2019).

[65] Dit, Bogdan, Revelle, Meghan, Gethers, Malcom, and Poshyvanyk, Denys. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process 25*, 1 (2013), 53–95.

[66] Durieux, Thomas, Madeiral, Fernanda, Martinez, Matias, and Abreu, Rui. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Tallinn, Estonia, 2019), pp. 302–313.

[67] Durieux, Thomas, Martinez, Matias, Monperrus, Martin, Sommerard, Romain, and Xuan, Jifeng. Automatic repair of real bugs: An experience report on the Defects4J dataset. *CoRR abs/1505.07002* (2015).

[68] Dwork, Cynthia, Kumar, Ravi, Naor, Moni, and Sivakumar, Dandapani. Rank aggregation methods for the web. In *International Conference on World Wide Web* (Hong Kong, 2001), pp. 613–622.

[69] Dwyer, Matthew B., Avrunin, George S., and Corbett, James C. Patterns in property specifications for finite-state verification. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (1999).

[70] Eaddy, Marc. Concern tagger case study data mapping the Rhino source code to the ECMA-262 specification). `http://www.cs.columbia.edu/~eaddy/concerntagger/`, 2007.

[71] Eaddy, Marc, Aho, Alfred V., Antoniol, Giuliano, and Guéhéneuc, Yann-Gaël. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *IEEE International Conference on Program Comprehension (ICPC)* (Amsterdam, The Netherlands, 2008), pp. 53–62.

[72] EclEmma. JaCoCo Java code coverage library. `https://www.eclemma.org/jacoco/`, 2017.

[73] Eclipse JDT API specification. `https://ibm.co/3orMarh`. [Online; accessed 4-March-2022].

[74] Engler, Dawson, Chen, David Yu, Hallem, Seth, Chou, Andy, and Chelf, Benjamin. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 57–72.

[75] Ernst, Michael D., Cockrell, Jake, Griswold, William G., and Notkin, David. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE) 27*, 2 (2001), 99–123.

[76] Essen, Christian Von, and Jobstmann, Barbara. Program repair without regret. *Formal Methods in System Design 47*, 1 (2015), 26–50.

[77] Evans, Robert B., and Savoia, Alberto. Differential testing: A new approach to change detection. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) Poster track* (Dubrovnik, Croatia, 2007), pp. 549–552.

[78] Fast, Ethan, Le Goues, Claire, Forrest, Stephanie, and Weimer, Westley. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference (GECCO)* (July 2010), pp. 965–972.

[79] Ferguson, Christopher J. An effect size primer: A guide for clinicians and researchers. *Professional Psychology: Research and Practice 40*, 5 (2009), 532–538.

[80] Forrest, Stephanie, Nguyen, ThanhVu, Weimer, Westley, and Le Goues, Claire. A genetic programming approach to automated software repair. In *Conference on Genetic and Evolutionary Computation (GECCO)* (Montreal, Québec, Canada, 2009), pp. 947–954.

[81] Fraser, Gordon, and Arcuri, Andrea. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE) 39*, 2 (February 2013), 276–291.

[82] Freire, Cibele, Gatterbauer, Wolfgang, Immerman, Neil, and Meliou, Alexandra. A characterization of the complexity of resilience and responsibility for self-join-free conjunctive queries. *VLDB Endowment (PVLDB) 9*, 3 (2015), 180–191.

[83] Fry, Zachary P., Landau, Bryan, and Weimer, Westley. A human study of patch maintainability. In *ISSTA* (Minneapolis, MN, USA, July 2012), pp. 177–187.

[84] Gabel, Mark, and Su, Zhendong. A study of the uniqueness of source code. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (Santa Fe, NM, USA, 2010), pp. 147–156.

[85] Galhotra, Sainyam, Brun, Yuriy, and Meliou, Alexandra. Fairness testing: Testing software for discrimination. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, September 2017), pp. 498–510.

[86] Gao, Xiang, Mechtaev, Sergey, and Roychoudhury, Abhik. Crash-avoiding program repair. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2019), pp. 8–18.

[87] Gay, Gregory, and Just, René. Defects4J as a challenge case for the search-based software engineering community. In *Proceedings of the International Symposium on Search-Based Software Engineering (SSBSE)* (Oct. 2020), pp. 255–261.

[88] Gazzola, L., Micucci, D., and Mariani, L. Automatic software repair: A survey. *IEEE Transactions on Software Engineering (TSE) 45*, 01 (Jan 2019), 34–67.

[89] Ghezzi, Carlo, Pezzè, Mauro, Sama, Michele, and Tamburrelli, Giordano. Mining behavior models from user-intensive web applications. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Hyderabad, India, 2014), pp. 277–287.

[90] Goffi, Alberto, Gorla, Alessandra, Ernst, Michael D., and Pezzè, Mauro. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis (ISSTA)* (Saarbrücken, Genmany, July 2016), pp. 213–224.

[91] Goodliffe, Pete. *Becoming a Better Programmer: A Handbook for People Who Care About Code.* " O'Reilly Media, Inc.", 2014.

[92] Gopinath, Divya, Malik, Muhammad Zubair, and Khurshid, Sarfraz. Specification-based program repair using SAT. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Saarbrücken, Germany, Mar. 2011), pp. 173–188.

[93] Griesmayer, Andreas, Bloem, Roderick, and Cook, Byron. Repair of boolean programs with an application to c. In *International Conference on Computer Aided Verification* (2006), Springer, pp. 358–371.

[94] Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. In *Symposium on Principles of Programming Languages (POPL)* (Austin, TX, USA, 2011), pp. 317–330.

[95] Gulwani, Sumit, Radiček, Ivan, and Zuleger, Florian. Automated clustering and program repair for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2018), pp. 465–480.

[96] Gupta, Rahul, Pal, Soham, Kanade, Aditya, and Shevade, Shirish K. DeepFix: Fixing common C language errors by deep learning. In *National Conference on Artificial Intelligence (AAAI)* (San Francisco, CA, USA, Feb. 2017), pp. 1345–1351.

[97] Harman, Mark. The current state and future of search based software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2007), pp. 342–357.

[98] Harman, Mark, and Jones, Bryan F. Search-based software engineering. *Information and Software Technology 43*, 14 (2001), 833–839.

[99] Harrold, Mary Jean, Rothermel, Gregg, Sayre, Kent, Wu, Rui, and Yi, Liu. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability 10*, 3 (2000), 171–194.

[100] Hill, Emily. Developing natural language-based program analyses and tools to expedite software maintenance. In *ICSE Doctoral Symposium* (2008), pp. 1015–1018.

[101] Hill, Emily, Rao, Shivani, and Kak, Avinash. On the use of stemming for concern location and bug localization in Java. In *SCAM* (2012), pp. 184–193.

[102] Hindle, Abram, Barr, Earl T., Su, Zhendong, Gabel, Mark, and Devanbu, Premkumar. On the naturalness of software. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2012), pp. 837–847.

[103] Hoffmann, Marc R., Janiczak, Brock, Mandrikov, Evgeny, and Friedenhagen, Mirko. JaCoCo code coverage tool. `https://www.jacoco.org/jacoco/`, 2009.

[104] Hua, Jinru, Zhang, Mengshi, Wang, Kaiyuan, and Khurshid, Sarfraz. Towards practical program repair with on-demand candidate generation. In *ACM/IEEE International Conference on Software Engineering* (June 2018), pp. 12–23.

[105] Ivanković, Marko, Petrović, Goran, Just, René, and Fraser, Gordon. Code coverage at Google. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Tallinn, Estonia, Aug. 2019), pp. 955–963.

[106] Jha, Susmit, Gulwani, Sumit, Seshia, Sanjit A., and Tiwari, Ashish. Oracle-guided component-based program synthesis. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2010), pp. 215–224.

[107] Jia, Yue, and Harman, Mark. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering 37*, 5 (2010), 649–678.

[108] Jiang, Jiajun. SimFix implementation. `https://github.com/xgdsmileboy/SimFix/`, 2017.

[109] Jiang, Jiajun, Wang, Ran, Xiong, Yingfei, Chen, Xiangping, and Zhang, Lu. Combining spectrum-based fault localization and statistical debugging: An empirical study. In *IEEE/ACM Int. Conf. on Automated Software Engineering* (2019), pp. 502–514.

[110] Jiang, Jiajun, Xiong, Yingfei, and Xia, Xin. A manual inspection of defects4j bugs and its implications for automatic program repair. *Science China Information Sciences 62*, 10 (2019), 200102.

[111] Jiang, Jiajun, Xiong, Yingfei, Zhang, Hongyu, Gao, Qing, and Chen, Xiangqun. Shaping program repair space with existing patches and similar code. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (Amsterdam, The Netherlands, July 2018), pp. 298–309.

[112] Jiang, Mingyue, Chena, Tsong Yueh, Kuoa, Fei-Ching, Toweyb, Dave, and Dingc, Zuohua. A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of Systems and Software 126* (April 2016), 127–140.

[113] Jiang, Nan, Lutellier, Thibaud, and Tan, Lin. Cure: Code-aware neural machine translation for automatic program repair. In *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), pp. 1161–1173.

[114] Jin, Guoliang, Song, Linhai, Zhang, Wei, Lu, Shan, and Liblit, Ben. Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2011), pp. 389–400.

[115] Jobstmann, Barbara, Griesmayer, Andreas, and Bloem, Roderick. Program repair as a game. In *International conference on computer aided verification* (2005), Springer, pp. 226–238.

[116] Jones, James A, and Harrold, Mary Jean. Empirical evaluation of the tarantula automatic fault-localization technique. In *IEEE/ACM Int. Conference on Automated Software Engineering* (2005), pp. 273–282.

[117] Jones, James A., Harrold, Mary Jean, and Stasko, John. Visualization of test information to assist fault localization. In *International Conference on Software Engineering (ICSE)* (Orlando, FL, USA, 2002), pp. 467–477.

[118] Jurafsky, Daniel, and Martin, James H. *Speech and Language Processing*, 2 ed. Pearson Education, Inc., 2009.

[119] Just, René, Jalali, Darioush, and Ernst, Michael D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA* (San Jose, CA, USA, July 2014), pp. 437–440.

[120] Just, René, Jalali, Darioush, Inozemtseva, Laura, Ernst, Michael D., Holmes, Reid, and Fraser, Gordon. Are mutants a valid substitute for real faults in software testing? In *ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China, 2014), p. 654–665.

[121] Just, René, Parnin, Chris, Drosos, Ian, and Ernst, Michael D. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In *International Symposium on Software Testing and Analysis (ISSTA)* (July 2018), pp. 287–297.

[122] Ke, Yalin, Stolee, Kathryn T., Le Goues, Claire, and Brun, Yuriy. Repairing programs with semantic code search. In *International Conference on Automated Software Engineering (ASE)* (Lincoln, NE, USA, November 2015), pp. 295–306.

[123] Kim, Dongsun, Nam, Jaechang, Song, Jaewoo, and Kim, Sunghun. Automatic patch generation learned from human-written patches. In *ICSE* (San Francisco, CA, USA, 2013), pp. 802–811.

[124] Kim, Sunghun, Zimmermann, Thomas, Whitehead Jr, E James, and Zeller, Andreas. Predicting faults from cached history. In *International Conference on Software Engineering* (2007), IEEE Computer Society, pp. 489–498.

[125] King, James C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (July 1976), 385–394.

[126] Kirbas, Serkan, Windels, Etienne, McBello, Olayori, Kells, Kevin, Pagano, Matthew, Szalanski, Rafal, Nowack, Vesna, Winter, Emily Rowan, Counsell, Steve, Bowes, David, Hall, Tracy, Haraldsson, Saemundur, and Woodward, John. On the introduction of automatic program repair in bloomberg. *IEEE Software 38*, 4 (2021), 43–51.

[127] Kneuss, Etienne, Koukoutos, Manos, and Kuncak, Viktor. Deductive program repair. In *International Conference on Computer Aided Verification* (2015), Springer, pp. 217–233.

[128] Knight, John C., and Leveson, Nancy G. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering (TSE) 12*, 1 (1986), 96–109.

[129] Ko, Andrew J, DeLine, Robert, and Venolia, Gina. Information needs in collocated software development teams. In *IEEE International Conference on Software Engineering (ICSE)* (2007), pp. 344–353.

[130] Kolde, Raivo, Laur, Sven, Adler, Priit, and Vilo, Jaak. Robust rank aggregation for gene list integration and meta-analysis. *Bioinformatics 28*, 4 (2012), 573–580.

[131] Koyuncu, Anil, Bissyandé, Tegawendé F, Kim, Dongsun, Liu, Kui, Klein, Jacques, Monperrus, Martin, and Traon, Yves Le. D&C: A divide-and-conquer approach to IR-based bug localization. *ArXiv abs/1902.02703* (2019).

[132] Koyuncu, Anil, Liu, Kui, Bissyandé, Tegawendé F, Kim, Dongsun, Monperrus, Martin, Klein, Jacques, and Le Traon, Yves. ifixr: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), ACM, pp. 314–325.

[133] Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, vol. 1. MIT Press, 1992.

[134] Krka, Ivo, Brun, Yuriy, Edwards, George, and Medvidovic, Nenad. Synthesizing partial component-level behavior models from system specifications. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Amsterdam, The Netherlands, August 2009), pp. 305–314.

[135] Kuo, Tzu-Ming, Lee, Ching-Pei, and Lin, Chih-Jen. Large-scale kernel ranksvm. In *Proceedings of the 2014 SIAM international conference on data mining* (2014), SIAM, pp. 812–820.

[136] Küçük, Yigit, Henderson, Tim A. D., and Podgurski, A. Improving fault localization by integrating value and predicate based causal inference techniques. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), 649–660.

[137] Lawrence, J, Clarke, Steven, Burnett, Margaret, and Rothermel, Gregg. How well do professional developers test with code coverage visualizations? an empirical study. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)* (2005), pp. 53–60.

[138] Le, Tien-Duy B., Le, Xuan Bach D., Lo, David, and Beschastnikh, Ivan. Synergizing specification miners through model fissions and fusions. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Lincoln, NE, USA, November 2015).

[139] Le, Tien-Duy B., and Lo, David. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015).

[140] Le, Tien-Duy B., Lo, David, Le Goues, Claire, and Grunske, Lars. A learning-to-rank based fault localization approach using likely invariants. In *International Symposium on Software Testing and Analysis* (2016), pp. 177–188.

[141] Le, Xuan Bach D., Bao, Lingfeng, Lo, David, Xia, Xin, Li, Shanping, and Pasareanu, Corina S. On reliability of patch correctness assessment. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (May 2019), pp. 524–535.

[142] Le, Xuan Bach D., Chu, Duc-Hiep, Lo, David, Le Goues, Claire, and Visser, Willem. JFIX: Semantics-based repair of Java programs via symbolic PathFinder. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (Santa Barbara, CA, USA, July 2017), pp. 376–379.

[143] Le, Xuan-Bach D., Chu, Duc-Hiep, Lo, David, Le Goues, Claire, and Visser, Willem. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, September 2017).

[144] Le, Xuan-Bach D, Le, Quang Loc, Lo, David, and Le Goues, Claire. Enhancing automated program repair with deductive verification. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2016), IEEE, pp. 428–432.

[145] Le, Xuan Bach D., Lo, David, and Le Goues, Claire. History driven program repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Mar. 2016), vol. 1, pp. 213–224.

[146] Le, Xuan Bach D., Thung, Ferdian, Lo, David, and Le Goues, Claire. Overfitting in semantics-based automated program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2018), pp. 163–163.

[147] Le Goues, Claire, Dewey-Vogt, Michael, Forrest, Stephanie, and Weimer, Westley. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *AMC/IEEE International Conference on Software Engineering (ICSE)* (Zurich, Switzerland, 2012), pp. 3–13.

[148] Le Goues, Claire, Forrest, Stephanie, and Weimer, Westley. Representations and operators for improving evolutionary software repair. In *Conference on Genetic and Evolutionary Computation (GECCO)* (July 2012), pp. 959–966.

[149] Le Goues, Claire, Holtschulte, Neal, Smith, Edward K., Brun, Yuriy, Devanbu, Premkumar, Forrest, Stephanie, and Weimer, Westley. The manybugs and introclass benchmarks for automated repair of C programs. *IEEE TSE 41*, 12 (December 2015), 1236–1256.

[150] Le Goues, Claire, Nguyen, ThanhVu, Forrest, Stephanie, and Weimer, Westley. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE) 38* (2012), 54–72.

[151] Le Goues, Claire, Pradel, Michael, and Roychoudhury, Abhik. Automated program repair. *Commun. ACM 62*, 12 (Nov. 2019), 56–65.

[152] Le Roy, Michael K. *Research Methods in Political Science: An Introduction Using MicroCase*, 7 ed. Wadsworth, Thompson Learning, 2009.

[153] Lee, Jaekwon, Kim, Dongsun, Bissyandé, Tegawendé F, Jung, Woosung, and Traon, Yves Le. Bench4BL: Reproducibility study on the performance of IR-based bug localization. In *International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands, 2018), pp. 61–72.

[154] Legunsen, Owolabi, Hassan, Wajih Ul, Xu, Xinyue, Roşu, Grigore, and Marinov, Darko. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *ASE* (2016), pp. 602–613.

[155] Li, Xia, Li, Wei, Zhang, Yuqun, and Zhang, Lingming. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), pp. 169–180.

[156] Li, Xia, Li, Wei, Zhang, Yuqun, and Zhang, Lingming. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. `https://github.com/DeepFL/DeepFaultLocalization`, 2019. [Online; accessed 4-March-2022].

[157] Li, Xia, and Zhang, Lingming. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (2017), 1–30.

[158] Li, Yi, Wang, Shaohua, and Nguyen, Tien. Fault localization with code coverage representation learning. In *IEEE/ACM International Conference on Software Engineering (ICSE)* (2021), pp. 661–673.

[159] Liblit, Ben, Naik, Mayur, Zheng, Alice X, Aiken, Alex, and Jordan, Michael I. Scalable statistical bug isolation. In *Acm Sigplan Notices* (2005), vol. 40, ACM, pp. 15–26.

[160] Lin, Shili. Rank aggregation methods. *Wiley Interdisciplinary Reviews: Computational Statistics 2*, 5 (2010), 555–570.

[161] Lin, Yiyan, and Kulkarni, Sandeep S. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In *International Symposium on Software Testing and Analysis (ISSTA)* (July 2014), pp. 237–247.

[162] Liu, Chen, Yang, Jinqiu, Tan, Lin, and Hafiz, Munawar. R2Fix: Automatically generating bug fixes from bug reports. In *IEEE International Conference on Software Testing, Verification and Validation* (2013), pp. 282–291.

[163] Liu, Kui, Kim, Dongsun, Koyuncu, Anil, Li, Li, Bissyandé, Tegawendé F., and Traon, Yves Le. A closer look at real-world patches. In *IEEE Int. Conference on Software Maintenance and Evolution (ICSME)* (2018), pp. 275–286.

[164] Liu, Kui, Koyuncu, Anil, Bissyandé, Tegawendé F, Kim, Dongsun, Klein, Jacques, and Traon, Yves Le. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *IEEE International Conference on Software Testing, Verification, and Validation* (Xian, China, 2019), pp. 102–113.

[165] Liu, Kui, Koyuncu, Anil, Kim, Dongsun, and Bissyandé, Tegawendé F. Tbar: Revisiting template-based automated program repair. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2019), pp. 31—-42.

[166] Liu, Kui, Li, Li, Koyuncu, Anil, Kim, Dongsun, Liu, Zhe, Klein, Jacques, and Bissyandé, Tegawendé F. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software 171* (2021), 110817.

[167] Liu, Peng, Tripp, Omer, and Zhang, Charles. Grail: Context-aware fixing of concurrency bugs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (Hong Kong, China, Nov. 2014), pp. 318–329.

[168] Liu, Xuliang, and Zhong, Hao. Mining StackOverflow for program repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Campobasso, Italy, Mar. 2018), pp. 118–129.

[169] Lo, David, and Khoo, Siau-Cheng. QUARK: Empirical assessment of automaton-based specification miners. In *Working Conference on Reverse Engineering (WCRE)* (2006).

[170] Lo, David, and Khoo, Siau-Cheng. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2006), pp. 265–275.

[171] Lo, David, and Maoz, Shahar. Scenario-based and value-based specification mining: Better together. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Antwerp, Belgium, 2010), pp. 387–396.

[172] Lo, David, Mariani, Leonardo, and Pezzè, Mauro. Automatic steering of behavioral model inference. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Amsterdam, The Netherlands, 2009), pp. 345–354.

[173] Logozzo, Francesco, and Ball, Thomas. Modular and verified automatic program repair. *ACM SIGPLAN Notices 47*, 10 (2012), 133–146.

[174] Long, Fan, Amidon, Peter, and Rinard, Martin. Automatic inference of code transforms for patch generation. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, Sept. 2017), pp. 727–739.

[175] Long, Fan, and Rinard, Martin. Staged program repair with condition synthesis. In *ESEC/FSE* (Bergamo, Italy, 2015), pp. 166–178.

[176] Long, Fan, and Rinard, Martin. An analysis of the search spaces for generate and validate patch generation systems. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Buenos Aires, Argentina, 2016), pp. 702–713.

[177] Long, Fan, and Rinard, Martin. Automatic patch generation by learning correct code. In *POPL* (St. Petersburg, FL, USA, 2016), pp. 298–312.

[178] Lou, Yiling, Ghanbari, Ali, Li, Xia, Zhang, Lingming, Zhang, Haotian, Hao, Dan, and Zhang, Lu. Can automated program repair refine fault localization? a unified debugging approach. In *ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA, 2020), pp. 75–87.

[179] Lutellier, Thibaud, Pham, Viet Hung, Pang, Lawrence, Li, Yitong, Wei, Moshi, and Tan, Lin. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020).

[180] Macho, Christian, McIntosh, Shane, and Pinzger, Martin. Automatically repairing dependency-related build breakage. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2018), pp. 106–117.

[181] Mahalakshmi, S, and Sivasankar, E. Cross domain sentiment analysis using different machine learning techniques. In *International Conference on Fuzzy and Neuro Computing (FANCCO)* (2015), pp. 77–87.

[182] Manning, Christopher D., Surdeanu, Mihai, Bauer, John, Finkel, Jenny, Bethard, Steven J., and McClosky, David. The Stanford CoreNLP natural language processing toolkit. In *ACL* (2014), pp. 55–60.

[183] Marginean, Alexandru, Bader, Johannes, Chandra, Satish, Harman, Mark, Jia, Yue, Mao, Ke, Mols, Alexander, and Scott, Andrew. SapFix: Automated end-to-end repair at scale. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Montreal, QC, Canada, May 2019), pp. 269–278.

[184] Martinez, Matias, Durieux, Thomas, Sommerard, Romain, Xuan, Jifeng, and Monperrus, Martin. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *EMSE 22*, 4 (April 2017), 1936–1964.

[185] Martinez, Matias, and Monperrus, Martin. ASTOR: A program repair library for Java (Demo). In *International Symposium on Software Testing and Analysis (ISSTA) Demo track* (Saarbrücken, Germany, 2016), pp. 441–444.

[186] Martinez, Matias, Weimer, Westley, and Monperrus, Martin. Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches. In *ACM/IEEE International Conference on Software Engineering New Ideas and Emerging Results track (ICSE NIER)* (Hyderabad, India, 2014), pp. 492–495.

[187] Matavire, Rangarirai, and Brown, Irwin. Profiling grounded theory approaches in information systems research. *European Journal of Information Systems 22*, 1 (2013), 119–129.

[188] Mechtaev, Sergey, Nguyen, Manh-Dung, Noller, Yannic, Grunske, Lars, and Roychoudhury, Abhik. Semantic program repair using a reference implementation. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, 2018), pp. 129–139.

[189] Mechtaev, Sergey, Yi, Jooyong, and Roychoudhury, Abhik. DirectFix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)* (Florence, Italy, May 2015), pp. 448–458.

[190] Mechtaev, Sergey, Yi, Jooyong, and Roychoudhury, Abhik. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)* (May 2016), pp. 691–701.

[191] Meliou, Alexandra, Gatterbauer, Wolfgang, Halpern, Joseph Y., Koch, Christoph, Moore, Katherine F., and Suciu, Dan. Causality in databases. *IEEE Data Engineering Bulletin 33*, 3 (2010), 59–67.

[192] Meliou, Alexandra, Gatterbauer, Wolfgang, Moore, Katherine F., and Suciu, Dan. The complexity of causality and responsibility for query answers and non-answers. *VLDB Endowment (PVLDB) 4*, 1 (2010), 34–45.

[193] Meliou, Alexandra, Gatterbauer, Wolfgang, and Suciu, Dan. Bringing provenance to its full potential using causal reasoning. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)* (2011).

[194] Meliou, Alexandra, Roy, Sudeepa, and Suciu, Dan. Causality and explanations in databases. *VLDB Endowment (PVLDB) tutorial 7*, 13 (2014), 1715–1716.

[195] Monperrus, Martin. A critical review of "Automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Hyderabad, India, June 2014), pp. 234–242.

[196] Moon, Seokhyeon, Kim, Yunho, Kim, Moonzoo, and Yoo, Shin. Ask the mutants: Mutating faulty programs for fault localization. In *IEEE International Conference on Software Testing, Verification and Validation* (2014), pp. 153–162.

[197] Motwani, Manish, and Brun, Yuriy. Automatically generating precise oracles from structured natural language specifications. In *International Conference on Software Engineering* (2019), IEEE Press, pp. 188–199.

[198] Motwani, Manish, and Brun, Yuriy. Automatically repairing programs using both tests and bug reports. *arXiv:2011.08340* (2020).

[199] Motwani, Manish, Sankaranarayanan, Sandhya, Just, René, and Brun, Yuriy. Do automated program repair techniques repair hard and important bugs? *EMSE 23*, 5 (October 2018), 2901–2947.

[200] Motwani, Manish, Soto, Mauricio, Brun, Yuriy, Just, René, and Le Goues, Claire. Quality of automated program repair on real-world defects. *IEEE Transactions on Software Engineering (TSE) 48*, 2 (Feb. 2022), 637–661.

[201] Muşlu, Kıvanç, Brun, Yuriy, and Meliou, Alexandra. Data debugging with continuous testing. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) NIER track* (Saint Petersburg, Russia, August 2013), pp. 631–634.

[202] Muşlu, Kıvanç, Brun, Yuriy, and Meliou, Alexandra. Preventing data errors with continuous testing. In *International Symposium on Software Testing and Analysis (ISSTA)* (Baltimore, MD, USA, July 2015), pp. 373–384.

[203] Newson, Roger. Parameters behind "nonparametric" statistics: Kendall's tau, Somers' D and median differences. *The Stata Journal 2*, 1 (2002), 45–64.

[204] Nguyen, Hoang Duong Thien, Qi, Dawei, Roychoudhury, Abhik, and Chandra, Satish. SemFix: Program repair via semantic analysis. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2013), pp. 772–781.

[205] Nimmer, Jeremy W., and Ernst, Michael D. Automatic generation of program specifications. In *International Symposium on Software Testing and Analysis (ISSTA)* (Rome, Italy, July 2002).

[206] Noda, Kunihiro, Nemoto, Yusuke, Hotta, Keisuke, Tanida, Hideo, and Kikuchi, Shinji. Experience report: How effective is automated program repair for industrial software? In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2020), pp. 612–616.

[207] Noller, Yannic, Shariffdeen, Ridwan, Gao, Xiang, and Roychoudhury, Abhik. Trust enhancement issues in program repair. In *IEEE/ACM International Conference on Software Engineering (ICSE)* (2022).

[208] Ohmann, Tony, Herzberg, Michael, Fiss, Sebastian, Halbert, Armand, Palyart, Marc, Beschastnikh, Ivan, and Brun, Yuriy. Behavioral Resource-Aware Model Inference. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (September 2014), pp. 19–30.

[209] Oliveira, Vinicius Paulo L., de Souza, Eduardo Faria, Le Goues, Claire, and Camilo-Junior, Celso G. Improved representation and genetic operators for linear genetic programming for automated program repair. *Empirical Software Engineering (EMSE) 23*, 5 (2018), 2980–3006.

[210] Oliveira, Vinicius Paulo L., Souza, Eduardo F. D., Le Goues, Claire, and Camilo-Junior, Celso G. Improved crossover operators for genetic programming for program repair. In *International Symposium on Search Based Software Engineering (SSBSE)* (2016), pp. 112–127.

[211] Pacheco, Carlos, and Ernst, Michael D. Randoop: Feedback-directed random testing for Java. In *Conference on Object-oriented Programming Systems and Applications (OOPSLA)* (Montreal, QC, Canada, 2007), pp. 815–816.

[212] Pacheco, Carlos, Lahiri, Shuvendu K., Ernst, Michael D., and Ball, Thomas. Feedback-directed random test generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (May 2007), pp. 75–84.

[213] Papadakis, Mike, and Le Traon, Yves. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability 25*, 5-7 (2015), 605–628.

[214] Parnin, Chris, and Orso, Alessandro. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis (ISSTA)* (Toronto, ON, Canada, 2011), pp. 199–209.

[215] Pearson, Spencer, Campos, José, Just, René, Fraser, Gordon, Abreu, Rui, Ernst, Michael D, Pang, Deric, and Keller, Benjamin. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering* (2017), IEEE Press, pp. 609–620.

[216] Pei, Yu, Furia, Carlo A., Nordio, Martin, Wei, Yi, Meyer, Bertrand, and Zeller, Andreas. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering (TSE) 40*, 5 (2014), 427–449.

[217] Perkins, Jeff H., Kim, Sunghun, Larsen, Sam, Amarasinghe, Saman, Bachrach, Jonathan, Carbin, Michael, Pacheco, Carlos, Sherwood, Frank, Sidiroglou, Stelios, Sullivan, Greg, Wong, Weng-Fai, Zibin, Yoav, Ernst, Michael D., and Rinard, Martin. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, October 12–14, 2009), pp. 87–102.

[218] Petke, Justyna, and Blot, Aymeric. Refining fitness functions in test-based program repair. In *IEEE/ACM International Conference on Software Engineering Workshops (ICSEW)* (202018), p. 13–14.

[219] Petke, Justyna, Haraldsson, Saemundur O., Harman, Mark, Langdon, William B., White, David R., and Woodward, John R. Genetic improvement of software: A comprehensive survey. *IEEE Transactions on Evolutionary Computation (TEVC) 22*, 3 (June 2018), 415–432.

[220] Pihur, Vasyl, Datta, Somnath, and Datta, Susmita. RankAggreg: Weighted rank aggregation. `https://cran.r-project.org/web/packages/RankAggreg/index.html`, 2020.

[221] Pihur, Vasyl, Datta, Susmita, and Datta, Somnath. RankAggreg, an R package for weighted rank aggregation. *BMC bioinformatics 10*, 1 (2009), 62.

[222] Polikarpova, Nadia, and Sergey, Ilya. Structuring the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–30.

[223] Posnett, Daryl, Filkov, Vladimir, and Devanbu, Premkumar. Ecological inference in empirical software engineering. In *International Conference on Automated Software Engineering (ASE)* (Lawrence, KS, USA, November 2011), pp. 362–371.

[224] Qi, Yuhua, Mao, Xiaoguang, and Lei, Yan. Efficient automated program repair through fault-recorded testing prioritization. In *ICSM* (Eindhoven, The Netherlands, Sept. 2013), pp. 180–189.

[225] Qi, Yuhua, Mao, Xiaoguang, Lei, Yan, Dai, Ziying, and Wang, Chengsong. The strength of random search on automated program repair. In *International Conference on Software Engineering (ICSE)* (2014), pp. 254–265.

[226] Qi, Yuhua, Mao, Xiaoguang, Lei, Yan, and Wang, Chengsong. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis (ISSTA)* (Lugano, Switzerland, July 2013), pp. 191–201.

[227] Qi, Zichao, Long, Fan, Achour, Sara, and Rinard, Martin. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA* (Baltimore, MD, USA, 2015), p. 24–36.

[228] Qiu, Xiaokang, and Solar-Lezama, Armando. Natural synthesis of provably-correct data-structure manipulations. *Proceedings of the ACM on Programming Languages 1*, OOPSLA (2017), 1–28.

[229] Rahman, Foyzur, Posnett, Daryl, Hindle, Abram, Barr, Earl, and Devanbu, Premkumar. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (Szeged, Hungary, 2011), ACM, pp. 322–331.

[230] Rahman, Md Masudur, Chakraborty, Saikat, Kaiser, Gail, and Ray, Baishakhi. A case study on the impact of similarity measure on information retrieval based software engineering tasks. *CoRR abs/1808.02911* (2018).

[231] Rational software architect 9.5.0. `https://ibm.co/3HfEml7`. [Online; accessed 4-March-2022].

[232] Reiss, Steven P. Semantics-based code search. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2009), pp. 243–253.

[233] Reiss, Steven P., and Renieris, Manos. Encoding program executions. In *ACM/IEEE Intl. Conference on Software Engineering* (2001), pp. 221–230.

[234] Renieres, Manos, and Reiss, Steven P. Fault localization with nearest neighbor queries. In *IEEE ASE* (2003), pp. 30–39.

[235] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. NIST Planning Report 02-3, May 2002.

[236] Robertson, Stephen, Zaragoza, Hugo, and Taylor, Michael. Simple BM25 extension to multiple weighted fields. In *CIKM* (2004), pp. 42–49.

[237] Robertson, Stephen E., Walker, Stephen, and Beaulieu, Micheline. Experimentation as a way of life: Okapi at TREC. *Information Processing and Management 36*, 1 (January 2000), 95–108.

[238] Robertson, Stephen E., Walker, Steve, and Beaulieu, M. Experimentation as a way of life: Okapi at trec. *Information processing & management 36*, 1 (2000), 95–108.

[239] Rolim, Reudismam, Soares, Gustavo, D'Antoni, Loris, Polozov, Oleksandr, Gulwani, Sumit, Gheyi, Rohit, Suzuki, Ryo, and Hartmann, Björn. Learning syntactic program transformations from examples. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (May 2017), pp. 404–415.

[240] Roychowdhury, Shounak, and Khurshid, Sarfraz. A novel framework for locating software faults using latent divergences. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2011), Springer, pp. 49–64.

[241] Roychowdhury, Shounak, and Khurshid, Sarfraz. Software fault localization using feature selection. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering* (2011), ACM, pp. 11–18.

[242] Roychowdhury, Shounak, and Khurshid, Sarfraz. A family of generalized entropies and its application to software fault localization. In *2012 6th IEEE International Conference Intelligent Systems* (2012), IEEE, pp. 368–373.

[243] Saha, Ripon K., Lease, Matthew, Khurshid, Sarfraz, and Perry, Dewayne E. Improving bug localization using structured information retrieval. In *IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)* (2013), pp. 345–355.

[244] Saha, Ripon K., Lyu, Yingjun, Yoshida, Hiroaki, and Prasad, Mukul R. ELIXIR: Effective object oriented program repair. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Nov. 2017), pp. 648–659.

[245] Samanta, Roopsha, Olivo, Oswaldo, and Emerson, E Allen. Cost-aware automatic program repair. In *International Static Analysis Symposium* (2014), Springer, pp. 268–284.

[246] Samar, Vipin, and Patni, Sangeeta. Differential testing for variational analyses: Experience from developing KConfigReader. *CoRR abs/1706.09357* (2017).

[247] Sanathanan, Lalitha. Estimating the size of a truncated sample. *Journal of the American Statistical Association 72*, 359 (1977), 669–672.

[248] Schur, Matthias, Roth, Andreas, and Zeller, Andreas. Mining behavior models from enterprise web applications. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Saint Petersburg, Russia, 2013), pp. 422–432.

[249] Scott, Andrew, Bader, Johannes, and Chandra, Satish. Getafix: Learning to fix bugs automatically. *CoRR abs/1902.06111* (2019).

[250] Sidiroglou, Stelios, and Keromytis, Angelos D. Countering network worms through automatic patch generation. *IEEE Security and Privacy 3*, 6 (Nov. 2005), 41–49.

[251] Smirnov, Alexey, and cker Chiueh, Tzi. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, USA, Feb. 2005).

[252] Smith, Edward K., Barr, Earl, Le Goues, Claire, and Brun, Yuriy. Is the cure worse than the disease? Overfitting in automated program repair. In *ESEC/FSE* (2015), pp. 532–543.

[253] Sobreira, Victor, Durieux, Thomas, Madeiral, Fernanda, Monperrus, Martin, and de Almeida Maia, Marcelo. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2018), pp. 130–140.

[254] softwaretestinghelp.com. 15 most popular bug tracking software to ease your defect management process. `http://www.softwaretestinghelp.com/popular-bug-tracking-software/`, accessed December 11, 2015.

[255] Sohn, Jeongju, and Yoo, Shin. FLUCCS: Using code and change metrics to improve fault localization. In *International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA, 2017), pp. 273–283.

[256] Soto, M., and Le Goues, C. Using a probabilistic model to predict bug fixes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (March 2018), pp. 221–231.

[257] Srivastava, Varun, Bond, Michael D., McKinley, Kathryn S., and Shmatikov, Vitaly. A security policy oracle: Detecting security holes using multiple API implementations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Jose, CA, USA, 2011), pp. 343–354.

[258] Steimann, Friedrich, Frenkel, Marcus, and Abreu, Rui. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *International Symposium on Software Testing and Analysis, (ISSTA)* (2013), pp. 314–324.

[259] Stolee, Kathryn T., and Elbaum, Sebastian. Toward semantic search via SMT solver. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) New Ideas and Emerging Results Track* (Cary, NC, USA, 2012), pp. 25:1–25:4.

[260] Stolee, Kathryn T., Elbaum, Sebastian, and Dobos, Daniel. Solving the search for source code. *ACM Transactions on Software Engineering Methodology 23*, 3 (May 2014), 26:1–26:45.

[261] Stolee, Kathryn T., Elbaum, Sebastian, and Dwyer, Matthew B. Code search with input/output queries: Generalizing, ranking, and assessment. *Journal of Systems and Software (JSS)* (2015).

[262] Strohman, Trevor, Metzler, Donald, HowardTurtle, and Croft, W. Bruce. Indri: A language model-based search engine for complex queries. In *International Conference on Intelligence Analysis* (2005), vol. 2, pp. 2–6.

[263] Sun, Shuyao, Guo, Junxia, Zhao, Ruilian, and Li, Zheng. Search-based efficient automated program repair using mutation and fault localization. In *Annual Computer Software and Applications Conference* (2018), vol. 1, pp. 174–183.

[264] Tan, Shin Hwei, Dong, Zhen, Gao, Xiang, and Roychoudhury, Abhik. Repairing crashes in android apps. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2018), pp. 187–198.

[265] Tan, Shin Hwei, Marinov, Darko, Tan, Lin, and Leavens, Gary T. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *International Conference on Software Testing, Verification, and Validation (ICST)* (Montreal, QC, Canada, 2012), pp. 260–269.

[266] Tan, Shin Hwei, Yi, Jooyong, Mechtaev, Sergey, and Roychoudhury, Abhik. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *IEEE International Conference on Software Engineering Poster Track* (May 2017), pp. 180–182.

[267] Tantithamthavorn, Chakkrit, Lemma, Surafel Abebe, Hassan, Ahmed E., Ihara, Akinori, and Matsumoto, Kenichi. The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology* (2018).

[268] Tian, Haoye, Liu, Kui, Kaboreé, Abdoul Kader, Koyuncu, Anil, Li, Li, Klein, Jacques, and Bissyandé, Tegawendé F. Evaluating representation learning of code changes for predicting patch correctness in program repair. *arXiv preprint arXiv:2008.02944* (2020), 981–992.

[269] Tian, Yuchi, and Ray, Baishakhi. Automatically diagnosing and repairing error handling bugs in C. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, Sept. 2017), pp. 752–762.

[270] Timperley, Christopher, Stepney, Susan, and Le Goues, Claire. Poster: BugZoo — A platform for studying software bugs. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, May 2018).

[271] Tu, Zhaopeng, Su, Zhendong, and Devanbu, Premkumar. On the localness of software. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (2014), pp. 269–280.

[272] Tufano, Michele, Watson, Cody, Bavota, Gabriele, Penta, Massimiliano Di, White, Martin, and Poshyvanyk, Denys. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol. 28*, 4 (sep 2019), 25–36.

[273] van Tonder, Rijnard, and Le Goues, Claire. Static automated program repair for heap properties. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2018), pp. 151–162.

[274] Walls, Robert J., Brun, Yuriy, Liberatore, Marc, and Levine, Brian Neil. Discovering Specification Violations in Networked Software Systems. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)* (Gaithersburg, MD, USA, November 2015), pp. 496–506.

[275] Wang, Ke, Singh, Rishabh, and Su, Zhendong. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, USA, June 2018), pp. 481–495.

[276] Wang, Qianqian, Brun, Yuriy, and Orso, Alessandro. Behavioral execution comparison: Are tests representative of field behavior? In *ICST* (2017), pp. 321–332.

[277] Wang, Shangwen, Wen, Ming, Lin, Bo, Wu, Hongjun, Qin, Yihao, Zou, Deqing, Mao, Xiaoguang, and Jin, Hai. Automated patch correctness assessment: How far are we? In *ACM International Conference on Automated Software Engineering (ASE)* (2020), Association for Computing Machinery, p. 968–980.

[278] Wang, Xiaolan, Dong, Xin Luna, and Meliou, Alexandra. Data X-Ray: A diagnostic tool for data errors. In *International Conference on Management of Data (SIGMOD)* (2015), pp. 1231–1245.

[279] Wang, Xiaolan, Meliou, Alexandra, and Wu, Eugene. QFix: Demonstrating error diagnosis in query histories. In *SIGMOD Demo* (2016), pp. 2177–2180.

[280] Wang, Xiaolan, Meliou, Alexandra, and Wu, Eugene. QFix: Diagnosing errors through query histories. In *SIGMOD* (2017), pp. 1369–1384.

[281] Weimer, Westley. Patches as better bug reports. In *Intl. Conference on Generative Programming and Component Engineering* (2006), pp. 181–190.

[282] Weimer, Westley, Fry, Zachary P., and Forrest, Stephanie. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE* (Palo Alto, CA, USA, 2013), pp. 356–366.

[283] Weimer, Westley, and Necula, George C. Finding and preventing run-time error handling mistakes. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2004), pp. 419–431.

[284] Weimer, Westley, Nguyen, ThanhVu, Le Goues, Claire, and Forrest, Stephanie. Automatically finding patches using genetic programming. In *ICSE* (Vancouver, BC, Canada, 2009), pp. 364–374.

[285] Weiss, Cathrin, Premraj, Rahul, Zimmermann, Thomas, and Zeller, Andreas. How long will it take to fix this bug? In *International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)* (2007), pp. 1–1.

[286] Wen, Ming, Chen, Junjie, Wu, Rongxin, Hao, Dan, and Cheung, Shing-Chi. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv preprint arXiv:1707.05172* (2017).

[287] Wen, Ming, Chen, Junjie, Wu, Rongxin, Hao, Dan, and Cheung, Shing-Chi. Context-aware patch generation for better automated program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, June 2018), pp. 1–11.

[288] Wen, Ming, Wu, Rongxin, and Cheung, Shing-Chi. Locus: Locating bugs from software changes. In *IEEE/ACM International Conference on Automated Software Engineering* (Singapore, 2016), pp. 262–273.

[289] White, Martin, Tufano, Michele, Martinez, Matias, Monperrus, Martin, and Poshyvanyk, Denys. Sorting and transforming program repair ingredients via deep learning code similarities. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), pp. 479–490.

[290] Wirfs-Brock, Allen, and Terlson, Brian. ECMA-262, ECMAScript 2017 language specification, 8th edition. `https://www.ecma-international.org/ecma-262/8.0`, 2017.

[291] Wong, Chu-Pan, Xiong, Yingfei, Zhang, Hongyu, Hao, Dan, Zhang, Lu, and Mei, Hong. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution* (Victoria, BC, 2014), IEEE, pp. 181–190.

[292] Wong, W Eric, Debroy, Vidroha, Gao, Ruizhi, and Li, Yihao. The DStar method for effective software fault localization. *IEEE Transactions on Reliability 63*, 1 (2013), 290–308.

[293] Wong, W. Eric, Gao, Ruizhi, Li, Yihao, Abreu, Rui, and Wotawa, Franz. A survey on software fault localization. *IEEE TSE 42*, 8 (2016), 707–740.

[294] Wu, Rongxin, Zhang, Hongyu, Cheung, Shing-Chi, and Kim, Sunghun. Crashlocator: locating crashing faults based on crash stacks. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)* (2014), pp. 204–214.

[295] Xie, Xiaoyuan, Chen, Tsong Yueh, Kuo, Fei-Ching, and Xu, Baowen. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM TOSEM 22*, 4 (2013), 31.

[296] Xin, Qi, and Reiss, Steven P. Identifying test-suite-overfitted patches through test case generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2017), pp. 226–236.

[297] Xiong, Yingfei, Liu, Xinyuan, Zeng, Muhan, Zhang, Lu, and Huang, Gang. Identifying patch correctness in test-based program repair. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)* (June 2018), pp. 789–799.

[298] Xiong, Yingfei, Wang, Jie, Yan, Runfa, Zhang, Jiachen, Han, Shi, Huang, Gang, and Zhang, Lu. Precise condition synthesis for program repair. In *ACM/IEEE International Conference on Software Engineering* (2017), pp. 416–426.

[299] Xuan, Jifeng, Martinez, Matias, Demarco, Favio, Clément, Maxime, Marcote, Sebastian Lamelas, Durieux, Thomas, Berre, Daniel Le, and Monperrus, Martin. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering (TSE) 43*, 1 (2016), 34–55.

[300] Xuan, Jifeng, Martinez, Matias, Demarco, Favio, Clement, Maxime, Marcote, Sebastian R. Lamelas, Durieux, Thomas, Berre, Daniel Le, and Monperrus, Martin. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering (TSE) 43*, 1 (2017), 34–55.

[301] Xuan, Jifeng, and Monperrus, Martin. Learning to combine multiple ranking metrics for fault localization. In *IEEE International Conference on Software Maintenance and Evolution* (Victoria, BC, 2014), pp. 191–200.

[302] Xuan, Jifeng, and Monperrus, Martin. Test case purification for improving fault localization. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China, 2014), pp. 52–63.

[303] Yang, Bo, He, Yuze, Liu, Huai, Chen, Yixin, and Jin, Zhi. A lightweight fault localization approach based on xgboost. In *IEEE International Conference on Software Quality, Reliability and Security (QRS)* (2020), pp. 168–179.

[304] Yang, Deheng, Qi, Yuhua, and Mao, Xiaoguang. Evaluating the strategies of statement selection in automated program repair. In *International Conference on Software Analysis, Testing, and Evolution* (2018), Springer, pp. 33–48.

[305] Yang, Jinqiu, Zhikhartsev, Alexey, Liu, Yuefei, and Tan, Lin. Better test cases for better automated program repair. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Paderborn, Germany, 2017), pp. 831–841.

[306] Yang, Xuejun, Chen, Yang, Eide, Eric, and Regehr, John. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2011), pp. 283–294.

[307] Ye, He, Martinez, Matias, Durieux, Thomas, and Monperrus, Martin. A comprehensive study of automatic program repair on the QuixBugs benchmark. In *IEEE International Workshop on Intelligent Bug Fixing* (Feb. 2019), pp. 1–10.

[308] Ye, He, Martinez, Matias, and Monperrus, Martin. Automated patch assessment for program repair at scale. *Empirical Software Engineering 26*, 2 (2021).

[309] Ye, Xin, Shen, Hui, Ma, Xiao, Bunescu, Razvan, and Liu, Chang. From word embeddings to document similarities for improved information retrieval in software engineering. In *international conference on software engineering (ICSE)* (2016), ACM, pp. 404–415.

[310] Youm, Klaus Changsun, Ahn, June, Kim, Jeongho, and Lee, Eunseok. Bug localization based on code change histories and bug reports. In *IEEE Asia-Pacific Software Engineering Conference* (New Delhi, India, 2015), pp. 190–197.

[311] Yu, Zhongxing, Martinez, Matias, Danglot, Benjamin, Durieux, Thomas, and Monperrus, Martin. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering 24*, 1 (Feb. 2019), 33–67.

[312] Yuan, Yuan, and Banzhaf, Wolfgang. ARJA. `https://github.com/yyxhdy/arja`, 2018. [Online; accessed 20-March-2022].

[313] Yuan, Yuan, and Banzhaf, Wolfgang. ARJA: Automated repair of Java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering 46*, 10 (oct 2020), 1040–1067.

[314] Zhai, Chengxiang, and Lafferty, John. A study of smoothing methods for language models applied to ad hoc information retrieval. In *SIGIR* (2001), pp. 334–342.

[315] Zhang, Lingming, Zhang, Lu, and Khurshid, Sarfraz. Injecting mechanical faults to localize developer faults for evolving software. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 765–784.

[316] Zhang, Mengshi, Li, Xia, Zhang, Lingming, and Khurshid, Sarfraz. Boosting spectrum-based fault localization using pagerank. In *ACM SIGSOFT Int. Symposium on Software Testing and Analysis (ISSTA* (2017), pp. 261–272.

[317] Zhang, Xiangyu, Gupta, Neelam, and Gupta, Rajiv. Locating faults through automated predicate switching. In *ACM ICSE* (2006), pp. 272–281.

[318] Zhong, Hao, and Su, Zhendong. An empirical study on real bug fixes. In *ACM/IEEE International Conference on Software Engineering* (May 2015).

[319] Zhou, Jian, Zhang, Hongyu, and Lo, David. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *IEEE International Conference on Software Engineering* (2012), pp. 14–24.

[320] Zou, Daming, Liang, Jingjing, Xiong, Yingfei, Ernst, Michael D, and Zhang, Lu. An empirical study of fault localization families and their combinations. *IEEE TSE* (2019).

[321] Zou, Daming, Liang, Jingjing, Xiong, Yingfei, Ernst, Michael D, and Zhang, Lu. Evaluating and combining fault localization techniques from different families. `https://damingz.github.io/combinefl/index.html`, 2019. [Online; accessed 4-March-2022].