Automated Program Transformation for Improving Software Quality

Rijnard van Tonder

CMU-ISR-19-101
October 2019

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Claire Le Goues, Chair
Christian Kästner
Jan Hoffmann
Manuel Fähndrich, Facebook, Inc.

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering.

Copyright 2019 Rijnard van Tonder

This work is partially supported under National Science Foundation grant numbers CCF-1750116 and CCF-1563797, and a Facebook Testing and Verification research award. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring corporation, institution, the U.S. government, or any other entity.

Keywords: syntax, transformation, parsers, rewriting, crash bucketing, fuzzing, bug triage, program transformation, automated bug fixing, automated program repair, separation logic, static analysis, program analysis

Abstract

Software bugs are not going away. Millions of dollars and thousands of developer-hours are spent finding bugs, debugging the root cause, writing a patch, and reviewing fixes. Automated techniques like static analysis and dynamic fuzz testing have a proven track record for cutting costs and improving software quality. More recently, advances in automated program repair have matured and see nascent adoption in industry. Despite the value of these approaches, automated techniques do not come for free: they must approximate, both theoretically and in the interest of practicality. For example, static analyzers suffer false positives, and automatically produced patches may be insufficiently precise to fix a bug. Such limitations continue to impose substantial human effort amid the benefits of automation.

Software development activities revolve around changing code. Thus, performing and reasoning about program change has extensive bearing on the effectiveness of automated techniques. From this perspective, we develop new *automated* techniques for changing programs to improve analysis behavior, and, correspondingly, use automated reasoning and analysis to specialize program changes for automated program repair. We present the first evidence that automated program transformation, program analysis, and program repair are interrelated and *cooperative*. We first show that automated program transformation leads to higher quality static analysis (by reducing false positives) and dynamic fuzz testing (by reducing duplicate bug reports). We then show how high-quality static analyses can feed into and enable automated program repair, and how automated repair can circle back to further improve static analysis (e.g., by revealing more true positive bugs). The thesis is that automated syntactic and semantic search and application of program transformations enables efficient, scalable, and unassisted techniques for improving the effectiveness of existing program analyses and end-to-end repair of real-world programs.

We show that these techniques are effective compared to current approaches in the respective domains of static analysis, dynamic fuzz testing, and program repair. We demonstrate relevance and real-world applicability by evaluating on large, popular, and active projects across multiple languages. Our vision for this work is that new capabilities and techniques for automated program transformation foster effective ways to automate burdensome human effort and reasoning incurred by limitations in program analysis and repair.

Acknowledgments

I'm incredibly grateful to my advisor, Claire, for taking me as her student. My first impression of Claire was that she would make a good advisor. Over time I realized that she's exceptional. I admire the focus and tact with which she's helped steer my work; always through keen questions and candid feedback. She's been my advocate and collaborator, and a valuable source of optimism. Our interactions had a special sense of levity that I'll always remember.

I thank my committee members, Christian, Jan, and Manuel. Their engagement was excellent and I greatly value their time and feedback. Their input has raised the quality of this dissertation.

I want to acknowledge the friendship and support of the following people during my studies: Arné, Nelius, Robert, Gerdus, Dominic, Nam, and Ivan. Then also those in the research group: Chris, Zack, Deby, Mau, Afsoon, Cody, and Jeremy.

I thank my family for their unceasing encouragement, prayers, and love: my father, Gerhard, my mother, Riana, and my two sisters, Monique and Karli. I'm especially grateful for the decisions and sacrifices that my parents made years and years ago—I realize now that those choices made it possible for me to accomplish what I have today.

None of what I accomplish matters without acknowledging Jesus Christ, who grants me all that I have.

"Now, for you: So what? The question I would leave you with is: Have you ever, as an adult, faced up to the question of whether the claims of Jesus Christ are true? And that's the most important question I can ask you."
— Fred Brooks, "Last Blast", 2015.

Contents

List of Terms **xv**

Preface **1**

1 Introduction **3**

1.1 Scope and Success Criteria 7

1.1.1 Lightweight Declarative Syntax Transformation 7

1.1.2 Tailoring Programs for Static Analysis 8

1.1.3 Tailoring Programs under Dynamic Fuzz Testing 9

1.1.4 Augmenting Static Analysis for Automated Program Repair 10

1.2 Contributions 11

1.3 Outline 13

2 Lightweight Declarative Syntax Transformation **15**

2.1 Introduction 16

2.2 Motivating Example 19

2.3 Dyck-Extended Languages 23

2.4 The Rewrite Process 26

2.4.1 Declarative Matching: The Intuition 26

2.4.2 Parser Parser Combinators for Matching 29

2.4.3 Match Rules 33

2.4.4 The Rewrite Phase . . . 34
2.5 Evaluation . . . 35
2.5.1 Real, Large Scale Multi-Language Rewriting . . . 37
2.5.2 Comparison with Existing Tools . . . 46
2.5.3 Discussion . . . 49
2.6 Related Work . . . 51
2.7 Summary . . . 53

3 Tailoring Programs for Static Analysis 55
3.1 Introduction . . . 55
3.2 Motivation . . . 59
3.3 Approach . . . 62
3.3.1 Match Template Behavior . . . 63
3.3.2 Tailoring Programs for Analysis . . . 63
3.4 Evaluation . . . 67
3.4.1 Experimental Setup . . . 69
3.4.2 Experimental Results . . . 71
3.4.3 Discussion . . . 82
3.5 Related Work . . . 84
3.6 Summary . . . 86

4 Semantic Crash Bucketing 87
4.1 Introduction . . . 87
4.2 Motivating Example . . . 91
4.3 Semantic Crash Bucketing . . . 94
4.3.1 Problem Formulation . . . 95
4.3.2 Detecting Duplicates . . . 97
4.3.3 Semantic Crash Bucketing Procedure . . . 98

4.4 Generating Approximate Fixes 100

4.4.1 $\widehat{T}$ Production. 100

4.4.2 Null Dereferences 101

4.4.3 Buffer Overflows 103

4.5 Experimental Design 105

4.5.1 Bugs with Ground Truth 106

4.5.2 Crash Corpus Generation 108

4.5.3 Evaluating Fuzzers 108

4.6 Experimental Results 111

4.6.1 Overall Results 111

4.6.2 Project-specific Results 113

4.7 Discussion 114

4.8 Related Work 116

4.9 Summary 118

5 End-to-End Static Automated Program Repair 119

5.1 Introduction 120

5.2 Preliminaries 125

5.2.1 Program Model and Assertion Language 126

5.2.2 Frame Inference 128

5.2.3 Finding Bugs Using Separation Logic 129

5.3 Repair with Separation Logic 130

5.3.1 Formulating Repair 131

5.3.2 Searching for Repairs 133

5.3.3 Applying Repairs: from Logic to Programs 136

5.4 Evaluation 140

5.4.1 Setup 141

5.4.2 Repair Results . . . 144
5.4.3 Patch Quality . . . 145
5.4.4 Fixing by Semantic Effects . . . 147
5.5 Limitations and Discussion . . . 149
5.6 Related Work . . . 150
5.7 Summary . . . 152

6 Conclusion and Future Work **153**
6.1 Summary . . . 153
6.2 Open Questions and Research Pursuits . . . 155
6.2.1 Extending Automated Program Repair to Additional Bug Classes . 156
6.2.2 Cooperative Automated Program Repair and Program Analysis . . 157
6.2.3 Inference in Place of Up-front Specification . . . 157
6.2.4 Automation and Integration with Developer Workflows . . . 158
6.2.5 Enriching Declarative Transformation with Semantic Information . 159

Bibliography **161**

List of Figures

2.1 A motivating description for declaratively rewriting redundant Go code. . . 20

2.2 An example application for simplifying Go code. 22

2.3 A base grammar for parsing multi-language syntax. 24

2.4 A visualization of matching parse trees. 27

2.5 A simple constraint grammar for match rules. 33

2.6 The match-rewrite templates used in large-scale experiments. 39

2.7 Parallelism yield when rewriting large programs. 45

3.1 A program change that works around a static analysis issue in `rsyslog`. . . 57

3.2 A motivating example of false positive reports in a PHP static analyzer. . . 60

3.3 Overview of the program tailoring process. 64

3.4 An example of program tailoring if-conditionals. 75

3.5 A program tailoring pattern for a substring in PHP. 76

3.6 A program tailoring pattern for free calls in C programs. 77

3.7 A program tailoring pattern for sockets in Java programs. 77

3.8 A program tailoring pattern for try-with-resource statements in Java. . . . 78

3.9 A program tailoring pattern for try-with statements in Java. 79

3.10 A program tailoring pattern for string compares in C programs. 80

3.11 A program tailoring pattern for string copies in C programs. 80

3.12 An example of tailoring C/C++ programs with buffer bounds checking. . . 81

3.13 A program tailoring pattern or `snprintf` functions in C programs. 82

4.1 Null dereference fixes in SQLite. 91

4.2 The Semantic Crash Bucketing procedure. 99

4.3 A rewrite template for fixing runtime crashes due to null dereferences. . . . 101

4.4 A rewrite template for bounding the size of `memcpy`. 104

5.1 Fixing a memory leak in the `Swoole` project 122

5.2 Fixing a null dereference in Google's error-prone tool. 123

5.3 The Smallfoot grammar and assertion language. 126

5.4 Modeling repair search. 134

5.6 Rewrite pattern for adding if-braces. 140

5.7 Repair Specifications. 141

5.8 Fixing resource leaks on multiple execution paths. 145

5.9 An example resource leak and fix. 147

List of Tables

2.1 Applying rewrite patterns at scale for 12 different languages. 36
2.2 Pull request results. 40
2.3 Rewrite tool comparison. 47

3.1 Main results of tailoring programs for static analysis. 68
3.2 Large scale search results for false positive-inducing patterns. 73
3.3 Summary of false positive issues in active analyzers. 74

4.1 Syntax extraction patterns for pointer dereferences. 102
4.2 Main results for the Semantic Crash Bucketing technique. 110

5.1 Main results for automated program repair using FootPatch. 143

List of Terms

abstract syntax tree An abstract syntax tree is a data structure that consists of program terms (e.g., expressions and statements) that represent a computation. Unlike a parse tree, an abstract syntax tree is a representation in which irrelevant syntax of the source code is removed, comprising only terms defined by a programming language's underlying grammar. 7, 15, 150

black box The term "black box", when used in reference to a process (as in a "black box analysis"), means that the inner workings, behavior, reasoning, and decisions of the process cannot be inspected or modified. In the context of our work, "black box" processes imply that we can only interact with the process by (a) providing input and (b) observing output of the process over time. 55

bug classes A bug in our context is a software flaw that leads to an error (i.e., undesirable program behavior); an error is a deviation from expected behavior defined by a validation oracle. A *bug class* refers to a sort of bug that shares an undesirable semantic trait. Bug classes occur across multiple programs in multiple locations. "Null dereference bugs" is a concrete example of a bug class. Analyses generally identify the set of bug classes they detect by name, or with an identifier (e.g., CWE-120 for security bugs). 6, 13, 56, 121, 132

context-free language Context-free languages and grammars describe properties of particular language expressivity in formal language theory. A context-free language is

one kind among four in Chomsky's hierarchy, being strictly more expressive than regular expressions, but strictly less expressive than context-sensitive languages. In the context of our work, context-free languages are most relevant in the sense that they express the property that a language exhibits well-formedness with respect to balanced delimiters (such as parentheses or braces). 8, 16

crash bucketing Crash bucketing is the process by which crashing program inputs are categorized into discrete groups, or buckets, where crashing inputs belonging to the same bucket are considered equivalent in that they trigger the same bug that induces a program crash. 10, 87

end-to-end And end-to-end process refers to the complete lifecycle of an automated process once it has been configured to run (e.g., possibly requiring some up-front specification). In this work, end-to-end repair repair refers to the complete lifecycle involving automated bug discovery followed by automated fix application and validation. 6, 11, 124

fuzzing Dynamic fuzz testing, or fuzzing, is an automated software testing technique that operates by mutating program inputs with the goal of crashing the program. Fuzzing comes in different flavors, and can involve coverage feedback to influence input mutation. 5, 10, 87

Hoare logic Hoare logic is a formal reasoning system that uses logical rules to prove correctness properties of programs. 125

parse tree A parse tree is a data structure that consists of program terms—it is a representation of source code that preserves all syntax. cf. abstract syntax tree . 15, 26

parser combinator A parser combinator is a function that parses syntax and typically outputs a program term (string literals, expressions, etc). Since parser combinators

are modeled as functions, they can be composed with higher-order functions (hence the word combinator) to implement a larger parser that implements a full grammar. 17

quality assurance Quality assurance in the context of this work is the process of evaluating and improving software quality using software tooling (via testing, analysis, and program changes). 12, 121

separation logic Separation logic is an extension of Hoare logic. It is a substructural logic that is particularly effective at modeling formal properties of heap memory by using the principle of local reasoning. 11

triage Triage is the process of checking the validity and severity of an incident as a precursor to remediating the issue. In the context of this work, we refer to the triage process of software bugs and crashing inputs, in particular distinguishing bug uniqueness in relation to crashing inputs. 9

unassisted An unassisted process means no human-in-the-loop is required for a process to complete. In the context of this dissertation a technique runs unassisted from the onset of a process, until the point where the final result or output is delivered (requiring no human intervention). Note that "unassisted" does not preclude one-off, up-front manual effort (e.g., specifying patch templates) which is done before the automated process begins. 6, 7, 153

validation oracle A validation oracle is a mechanism that defines a correctness property, and can check whether an input satisfies that correctness property. In the context of this work, inputs are programs, where we are interested in whether a program passes a validation oracle (which may be defined by a particular semantic property or test). 89, 95

Preface

This dissertation is a compilation and synthesis of four major papers [196, 197, 199, 200]. The structure and presentation is enriched to promote the overall thesis. Additions include extended discussion of techniques, results, and applications, as well as follow-on research pursuits of this work.

- i Rijnard van Tonder and Claire Le Goues. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Programming Language Design and Implementation*, PLDI '19, pages 363–378, 2019
- ii Rijnard van Tonder and Claire Le Goues. Tailoring Programs for Static Analysis via Program Transformation *(under submission)*, 2019
- iii Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic Crash Bucketing. In *Automated Software Engineering*, ASE '18, pages 612–622, 2018
- iv Rijnard van Tonder and Claire Le Goues. Static Automated Program Repair for Heap Properties. In *International Conference on Software Engineering*, ICSE '18, pages 151–162, 2018. (ACM Distinguished Paper Award)

Chapter 1

Introduction

Software bugs are not going away. Millions of dollars and thousands of developer-hours are spent finding bugs, debugging the root cause, writing a patch, and reviewing fixes [56, 167]. Automated tools such as linters, static analyzers, and fuzzers help drive down the costs by catching bugs early and increase developer efficiency [43, 47, 61, 83]. Recent techniques in automatic program repair [106, 136, 158] seek to further cut costs by automatically and correctly fixing bugs. Yet automated techniques do not come for free; they must approximate, both theoretically [83, 129, 142] and in the interest of practicality [61, 68, 104]. For example, static analyzers suffer false positives [172], and automatically produced patches may be insufficiently precise to fix a bug [190]. Such limitations impose substantial human effort and reasoning to realize benefit (e.g., a developer needs to discern true versus false positive bug reports, or judge the correctness of a patch). Thus, despite the proven value of automated techniques for improving software quality [61, 83, 151, 186], abstraction tradeoffs in automated reasoning, design, and implementation lead to imprecision that incurs significant manual effort, hindering broad adoption and utility [67].

Developing and maintaining software centers around a core activity: making changes to programs [149]. Software development is a complex and dynamic process owing to the fact that programs are highly subject to change due to bug fixes, refactoring, new features, or

changes in the underlying language syntax or runtime [123, 161, 192]. Automated techniques for manipulating, analyzing, and repairing programs offer efficient means (i.e., compared to manual human effort) for developing and assuring software quality particularly as programs change [61, 83]. Program changes in turn affect the applicability, correctness, or behavior of refactoring tools, analyzers, and autofix tools. *Performing and reasoning about program change thus has extensive bearing on the effectiveness of automated techniques.* Where predominantly manual effort intervenes to overcome limitations in automated techniques (e.g., filtering false positives or fixing bugs) we ask: How might better techniques for *automatically* performing and reasoning about program changes improve the effectiveness of automated techniques for ensuring software quality? This dissertation investigates interrelated problem domains where limitations in automated techniques give way to manual effort, and shows that automated program transformation can address them in new and effective ways.

We first consider static analyzers. While effective, static analyzers routinely report false positive bugs. Causes range from imprecision in the analysis [68] to semantic differences across different language runtime versions. Existing mechanisms for suppressing warnings prove insufficiently granular [67, 116]. Interestingly, developers instinctively recognize that they can change their programs in small ways to remove false positives.[1] This alternative is however manual and ad hoc. In general, the inability to easily and selectively perform non-trivial program transformations, automatically at scale, diminishes the benefits that could hypothetically be derived through generic and broad application.

This observation bears on a second open problem domain: performing precise crash triaging. Large scale dynamic fuzz testing can report hundreds of unique crash reports that correspond to the same bug, even after using deduplication heuristics [66, 76]. Like false positive bug reports, duplicate fuzzer reports can incur considerable human effort to filter. Humans predominantly debug crash-inducing behavior to ultimately *produce a fix* (i.e., a

[1] See, e.g., https://github.com/phpstan/phpstan/issues/1739#issuecomment-450582305

program change that positively alters behavior). This process suggests the possibility of automatically applying program transformations to silence or suppress inputs that otherwise produce duplicate bug reports in current techniques.

Third and finally, automatic program repair seeks to *actually* fix bugs. To become mainstream, automated program repair must produce *semantically* correct patches that can be trusted for automatic application. The vast majority of automated program repair research relies on existing tests to identify bugs, validate fixes, or both [120, 135, 135, 145, 146, 154, 164, 177, 211]. One fundamental limitation of tests is that they are limited to testing *known* functionality of programs, and hence cannot be used to fix newly discovered bugs automatically. To truly propel real-world adoption, automated program repair must expand to bugs discovered by *automated* techniques like static analysis and fuzzing. Static analysis and verification provides automated reasoning for bug-finding, yet automated program repair research based on analysis tooling remains underexplored with limited practical applicability to real-world software [115, 126, 202]. Recent advances deliver high quality analysis techniques and tools to cost effectively find real bugs [61]. This enables the exploration of (a) efficient semantic-driven search and application of program transformation and (b) approaches that provide high confidence in patch correctness by validating changes with respect to the semantic analysis domain.

The problem domains we consider in this dissertation (concretely, static analysis, dynamic fuzz testing, and program repair) are generally pursued separately in the literature.[2] In this dissertation we principally show that new combinations of automated reasoning and program transformation can broadly improve the effectiveness of techniques in these domains respectively. Further, our solutions present the first empirical evidence of another key insight: automated program transformation across these domains also create *cooperative* solutions, as these domains are interrelated. At a high level, we achieve an effective automated

[2]As are related, but distinct areas in verification, symbolic execution, testing, and so on, which we do not consider deeply.

program repair technique by building on top of a static analysis. Thus, improving the static analysis (in our case, via program transformation) can yield an enhanced automated program repair procedure. In the other direction, automated program repair techniques offer the auxiliary ability (i.e., beyond fixing bugs) to also amplify the effectiveness of static analysis (by uncovering more true positive bugs) and dynamic fuzz testing (by providing an oracle to distinguish unique crashing inputs).

In short, we pose that specializing automated program transformation presents new and effective ways to improve automated reasoning and analysis (1), and correspondingly, that using automated reasoning and analysis to specialize automated program transformation enables new and effective ways to achieve automated program repair (2). Concretely, we pursue open problem domains in (1) to reduce false positives reported by static analyzers and duplicate bug reports by dynamic fuzzers, and automated repair of automatically discovered bugs in (2). We consider that our solutions across these domains have a cooperate effect. In the one direction, removing false positives can improve the effectiveness of automated program repair. In the other, automated program repair can amplify the effectiveness of static analysis and dynamic fuzz testing. Our solutions entail both syntactic and semantic reasoning to drive program transformations, which we apply to real-world software. Thus:

The thesis is that automated syntactic and semantic search and application of program transformations enables efficient, scalable, and unassisted techniques for improving the effectiveness of existing program analyses and end-to-end repair of real-world programs.

We document the invention, design, and implementation of automated techniques that improve existing analyses in static and dynamic settings, as well as the first unassisted, end-to-end automated program repair system for multiple bug classes, and demonstrate their effectiveness on real programs. Section 1.1 gives a brief description of the techniques that substantiate the thesis, their scope, and criteria for success. Section 1.2 summarizes

our contributions, and Section 1.3 outlines the rest of the dissertation.

1.1 Scope and Success Criteria

The techniques in this dissertation emphasize applicability to modern real-world programs. We evaluate techniques on actively used software and tooling for multiple languages. All techniques are shown to *scale* to large software projects (>100KLOC). We demonstrate new applications of automated program transformation to problem domains that have not been attempted before. We thus characterize *efficiency* and *effectiveness* with respect to these domains for each application in the following sections. All techniques run *unassisted*, meaning no human-in-the-loop is required for an automated procedure to complete. An unassisted process does not preclude up-front manual effort (e.g., defining templates that rewrite program fragments), which is generally required to initiate the automated procedures in our work. While up-front effort is manual, our requirements are lightweight and serve an important role in allowing developers to configure and parameterize automated program transformation and reasoning. Developers are sensitive to whether tools can integrate into existing workflows [116]. Our techniques accomodate this view by allowing domain experts to integrate specialist knowledge in targeted transformations and automated actions. Section 6 elaborates on further work to automate inference of up-front specifications.

1.1.1 Lightweight Declarative Syntax Transformation

The ability to easily and automatically change programs lies on the critical path to efficiently improving software using program transformation. Current solutions for changing programs generally requires interfacing with complex, language-specific abstract syntax tree (AST) frameworks that fail to generalize across multiple languages in cases where we simply want to make small syntactic changes. Notably, while our repair technique (Chapter 5) was not purely based on syntactic changes, we found that manipulating syntax (as motivated by

underlying semantic reasoning) presented a limiting factor to performing changes. This problem was exacerbated in attempts to tailor programs for analyses.

Our solution is a declarative approach to syntax transformation for multiple languages. The goal in supporting multiple languages is to enable subsequent approaches in analysis tailoring and repair to generalize in a language-agnostic ways. Transformation across multiple languages is hard due to heterogeneous representations in syntax, parse trees, and abstract syntax trees. Our approach decomposes this problem to a common grammar expressing central context-free language properties shared by modern languages (e.g., balanced parentheses), while using smaller parsers to handle syntactic differences across languages (e.g., comments). We operationalize this decomposition using a parser-generating parser using parser combinators. We call such a parser a Parser Parser Combinator, which generates parsers for matching syntactic fragments in source code by parsing declarative user-supplied templates. Modulo up-front specification of templates for matching and rewriting programs, the procedure runs unassisted. We show that our approach meets efficiency and scalability criteria, in addition to generality: program transformation is comparatively fast (i.e., efficient) compared to language-specific tooling, effective across 12 languages, and scales to hundreds of millions of lines of code. This approach enables a fundamental primitive for efficient, scalable, and unassisted syntactic search and application of program transformations.

1.1.2 Tailoring Programs for Static Analysis

Static analysis is a proven technique for ensuring software quality, but tools approximate in the interest of theoretical and practical limits. False positives are a pervading manifestation of such approximations. To suppress false positives, developers readily disable bug checks or insert comments that suppress spurious bug reports. Existing work shows that these mechanisms fall short of developer needs and present a significant pain point for using or

adopting analyses [67].

Our solution applies targeted program transformations to code patterns that induce false positives. We implement our solution using declarative templates (as in Section 1.1.1). We make the observation that analysis users have little agency over the format of analysis configuration options provided to them, but that program transformation offers a fresh primitive for leveraging influence over analysis behavior.

The underlying hypothesis is that rule-based application of program transformations can improve the effectiveness static analysis by reducing false positive bug reports. The notion that syntactic transformations abstract semantic transformations [75] underpins our intuition that manipulating syntax can achieve desirable changes in the analysis domain and implementation.

A key objective of our work is to demonstrate the broad feasibility and effectiveness of these ideas for the first time in practice. To be feasible, program tailoring must be efficient compared to running the analysis. To be effective, it must improve analysis reports: i.e., remove bona fide false positives without adversely affecting results otherwise. We demonstrate this efficiency and effectiveness across multiple analyzers (e.g., Clang, SpotBugs, and PHPStan) and languages (C, Java, and PHP). Further, we show that our technique can be used at scale to even automatically detect and remedy false positive patterns before they are surfaced by an analyzer.

1.1.3 Tailoring Programs under Dynamic Fuzz Testing

Precise crash triage is important for automated dynamic testing tools, like fuzzers. At scale, fuzzers produce millions of crashing inputs. Fuzzers use heuristics, like stack hashes, to cut down on duplicate bug reports. These heuristics are fast, but often imprecise: even after deduplication, hundreds of uniquely reported crashes can still correspond to the same bug [66, 78]. Remaining crashes must be inspected manually, incurring considerable effort.

Analogous to our solution for false positives in static analyzers, we design a generic method for precise crash bucketing using program transformation. We call this method Semantic Crash Bucketing, which maps crashing inputs to unique bugs as a function of changing a program (i.e., a semantic delta). We observe that a real bug fix precisely identifies crashes belonging to the same bug. The insight is to approximate real bug fixes with lightweight program transformation to obtain the same level of precision.

The underlying hypothesis is that rule-based application of program transformations can improve fuzzing by reducing the number of duplicate bug reports compared to existing techniques. As in Section 1.1.2, syntactic patch templates effect favorable semantic changes toward our goal of reducing duplicate crash reports. To better approximate a fix in this dynamic setting, we additionally incorporate semantic feedback from program execution traces to inform patching.

As in Section 1.1.2, our approach must be efficient and effective compared to existing approaches, but now in the domain of fuzz testing and crash deduplication. We show that patch generation and deduplication is fast (on the order of tens of seconds) up to very large programs. We demonstrate effectiveness by showing that approximate fixes are competitive with ground truth fixes for crash deduplication, and significantly outperforms built-in deduplication techniques for three state-of-the-art fuzzers.

1.1.4 Augmenting Static Analysis for Automated Program Repair

Static analysis tools have demonstrated effectiveness at finding bugs in real-world code. Automatic Program Repair has the potential to further cut costs by automatically fixing bugs. Moreover, *real* bug fixes also deliver new ways to positively influence analysis behavior, as in Section 1.1.3. However, there is a disconnect between effective bug-finding tools and the ability to then fix bugs with Automated Program Repair (APR). Recent APR advances largely rely on test cases [120, 135, 135, 145, 146, 154, 164, 177, 211], making

them inapplicable for fixing newly discovered bugs found by analyzers.

Our solution adapts advances in practical static analysis and verification techniques to enable a new technique that finds and then accurately fixes real bugs without tests. To guard against patch overfitting [190], we build on an underlying semantic domain (using separation logic) to inform program transformation. At a high level, our technique reasons over semantic effects of existing program fragments to fix faults related to general pointer safety properties: resource leaks, memory leaks, and null dereferences. Thus, like Section 1.1.3, we use an appropriate semantic basis to drive automated patching (in this case, afforded by a static analysis). In the setting of automated program repair, this work focuses on producing real bug fixes, including ones for previously undiscovered bugs. Our approach uses the static analyzer's reasoning to provide a correctness criterion in lieu of tests: bug fixes correct the residual error state detected in the separation logic domain. The underlying hypothesis is that static analysis using the domain of separation logic enables rule-based semantic search and application of program transformations to fix real-world programs.

We implement our approach in a tool called FootPatch, an extension of the Infer static analyzer [5], to enable an end-to-end approach for detecting and fixing bugs. FootPatch is efficient, and can generate fixes for large programs (>100KLOC) in under 5 minutes. We show that FootPatch is effective: it generates correct patches in multiple languages (C and Java) for previously-unknown bugs, which have been merged upstream into highly popular projects. We also show anecdotal evidence that fixing bugs allows the static analysis to subsequently discover more true positive bugs.

1.2 Contributions

The main contributions of this dissertation are as follows.

1. A new technique for declaratively and selectively performing program transformations

(Chapter 2). We formulate a modular syntax rewriting technique that generalizes to multiple languages and scales to hundreds of millions of lines of code.

2. A new technique introducing the idea of tailoring programs for static analysis using program transformation (Chapter 3). We use declarative templates from (1) to enable a new mechanism that detects and suppresses false positives introduced by long-standing, unaddressed, and complex issues in actively used analyzers.

3. A new technique for tailoring programs under dynamic fuzz testing using program transformation (Chapter 4). We develop a generic technique for precisely deduplicating crash reports as a function of changing a program (i.e., a semantic delta). Our approach outperforms built-in deduplication techniques for state-of-the-art fuzzers.

4. An automatic program repair technique that uses separation logic and static analysis to drive program transformations that fully fix real-world bugs (Chapter 5). Our approach produced patches that are merged into active upstream projects, marking an important milestone in end-to-end repair by correctly and automatically fixing previously undiscovered bugs.

The contributions of this dissertation emphasize real-world applicability to existing programs and software quality assurance techniques. The following is a high level summary of the empirical contributions corresponding to the above, in substantiating the validity of our claims.

- Efficient and general syntactic program transformation, evaluated on the 100 most popular GitHub projects for each of 12 different languages, comprising 282 million lines of code. Our approach has led to more than 50 automatically produced patches that have been merged into over 40 of the most popular open source projects on GitHub, including the Rust compiler.[3]
- An evaluation on five actively used static analyzers for three different languages

[3]Broadly, we produce syntactic changes that improve code readability, style, and performance.

(C, Java, PHP). We show a reduction of 126 false positive reports across 15 large real-world projects (including OpenSSL, 400KLOC in size) for nine different bug classes using tailored program transformation.

- A comparative evaluation of our crash bucketing approach and five other crash deduplication strategies implemented in three state-of-the-art fuzzers. We evaluate on crash reports related to 21 real-world bugs (belonging to two bug classes: null dereferences and buffer overflows) in six large projects, including the PHP interpreter and SQLite. We show that our approach, using program transformation, can detect tens to hundreds of duplicate crashing inputs where fuzzer techniques cannot.
- Automatic patch generation for semantic bug fixing. We generate 55 total bug fixes using our program repair approach across 11 large projects for three bug classes (null dereferences, resource leaks, and memory leaks) across two languages (C and Java). Six of our fixes are merged upstream into popular C and Java projects.

1.3 Outline

Chapter 2 introduces our technique for declaratively rewriting programs. All subsequent work in Chapters 3, 4, and 5 use this technique in some capacity. Chapter 3 describes program tailoring via transformation for static analysis. Chapter 4 transitions to program tailoring in a dynamic setting (i.e., fuzz testing), where the objective focuses on deduplicating runtime crash reports. Chapter 5 explains our approach augmenting an existing bug-finding analysis to inform end-to-end program repair. Since our work in program transformation cuts across orthogonal objectives in static analysis, dynamic testing, and program repair, we discuss related work of existing techniques separately at the end of each respective chapter. Chapter 6 discusses future work and concludes.

Chapter 2

Lightweight Declarative Syntax Transformation

This chapter presents a new technique for declaratively manipulating program syntax. The motivation is that program transformation primitives enable new and effective ways for improving software (e.g., via refactoring and automated program repair). To harness this potential at scale, we need simple, efficient, and general ways of performing selective and automated program transformation. Generalizing to multiple language is especially hard due to heterogeneous representations in syntax, parse trees, and abstract syntax trees. This chapter presents a new technique and tool that achieves these goals: efficient, declarative syntax manipulation that generalizes to multiple languages. We demonstrate these ideas across 12 languages, and validate effectiveness of our approach by producing correct and desirable lightweight transformations on popular real-world projects. The capabilities of this technique are used in some capacity in all subsequent chapters.

2.1 Introduction

Automatically transforming programs is hard, yet critical for automated program refactoring [26, 29, 155], rewriting [20, 150], and repair [121, 145, 191, 197]. The complexity of automatically transforming code has yielded a plethora of approaches and tools that strike different design points in expressivity, language-specificity, interfaces, and sophistication [26, 27, 31, 110, 150]. At one end of the spectrum, techniques based on regular expressions, like `sed` or `codemod` [27], perform simple textual substitutions. On the other, language-specific AST visitor frameworks, like `clang-reformat`, manipulate rich program structures and metadata. Unfortunately, many lightweight transformations that should apply to multiple languages are entangled in language-specific tools with complex interfaces, relying on a program's AST to work. For example, consider a simple list slicing "quick fix": both Python and Go can be simplified from α`[`β`:len(`α`)]` to α`[`β`:]` for a list assigned to variable α and sliced from initial index β. Visually, the transformation is lightweight and easy to describe. Unfortunately, *actually performing* this type of transformation generally boils down to separate implementations depending on language-specific ASTs.

We observe that a wide range of lightweight transformations rely on matching tokens structurally within well-formed delimiters (essential syntax exhibiting a context-free property, and common to virtually all languages [28, 29, 31, 34, 37, 38]).[1] We propose an approach that exploits this observation to enable short (yet nontrivial) rewrite specifications that elide the need for complex, language-specific implementations, syntax definitions, or translations from metasyntax to an underlying abstract representation. We introduce a modular grammar that can be modified based on syntactic differences in languages (e.g., comment syntax), but which preserves the quintessential property of context-free languages. Our grammar extends Dyck languages [53], which capture the CFL constructs of interest in conventional

[1]Note that valid programs in many languages exhibit additional *context-sensitive* language features (e.g., parsing valid type definitions and their uses in C). Our approach does not attempt to parse strictly valid programs for a language, but rather provides sufficient expressivity to parse context-free properties found in language syntax.

programming languages. We develop a Dyck-Extended Language (DEL) grammar that can interpolate tokens (i.e., regular strings) between and inside balanced delimiters as a base representation for syntactic manipulation.

The crux of our approach implements a parser generator for a DEL; the generated parser matches concrete syntax of interest, enabling rewriting. Importantly, the parser generator is modular (reflecting the fact that DEL grammars are modularly defined), which we implement using parser combinators. We refer to our modular parser generators as "Parser Parser Combinators", or PPCs. Thus, the parser generator contains open extension points where smaller parsers handle language specific syntax (e.g., custom delimiters and comment syntax) in the generated match parser it produces.

The first advantage of using PPCs is that our approach does not rely on any source code translation at all. It instead detaches from representing input programs with any particular abstract syntax tree representation: the original source input *is* our concrete representation, and matching syntax relies only on how a parser interprets that input. Traditional approaches rely on first parsing syntax into a tree representation which is then manipulated. By contrast, our approach embeds the structural properties of syntax in the generated parsers alone; the parser records only matching syntax during parsing. PPCs enable easy multi-language extension for varying syntax across languages by exposing hooks for smaller parsers in a skeleton parser. Extending the parser to language-specific syntax (e.g., for comments) in the parsing routine is comparatively easy in our architecture as compared to modifying concrete AST definitions in code.

The second advantage of PPCs is that they interpret user-specified match templates *as* parser generators for a DEL. Small, custom syntactic parsers embed into the PPC to parameterize both how user-supplied templates are interpreted, *and* the parsers that those templates produce. The effect is that we avoid the complexity of implementing an additional translation layer from user-supplied metasyntax to an abstract syntax representation; the same custom syntax definitions embed seamlessly into the parser for user-supplied templates

and the match procedure (a parser) produced from it. The result is rewrite templates that are syntactically close to the target patterns with minimal metasyntax.

The main contributions of this chapter are:

- The Dyck-Extended Language representation, defined by a modular grammar for accommodating syntactic idiosyncrasies and ambiguity across multiple languages. We operationalize extensible DEL grammars using:
- Parser Parser Combinators for performing lightweight syntactic rewriting. PPCs enable an extensible parser-generating procedure that operates directly on source code without the need for an intermediate data structure. This enables:
- Declarative specification of rewrite templates (syntactically close to the target syntax) that are interpreted as parser definitions, eliding the need for additional translation layers converting the user-facing templates to an underlying representation and vice-versa.
- A large-scale empirical evaluation to demonstrate the effectiveness of our lightweight approach.

The evaluation aligns with the goals of the thesis: to present a new way of enabling efficient syntax matching and transformation that scales generally across multiple languages and large programs. These attributes further enable efficient automated techniques in subsequent chapters.

We thus evaluate our approach on 12 languages comprising 185 million lines of code using 30 transformations. We ran our transformations on the 100 most popular GitHub[2] repositories for each language. By targeting actively developed projects, we had the opportunity to submit our changes upstream; our approach has produced over 50 syntactic changes that are merged into over 40 of the most popular open source projects on GitHub. Merged changes reflect an important additional validation that our transformation engine

[2] In October, 2018.

produces correct and desirable syntactic changes. We also compare our multi-language approach to nine checkers and rewriters for individual languages. We demonstrate declarative specifications (typically less than 10 lines each) for existing real-world transformations that otherwise require tens of lines of implemented in different languages in current real-world tools.

The tool and rewrite specifications are available online[3] as well as 100 real-world example transformations produced by our work.[4]

2.2 Motivating Example

The `staticcheck`[5] tool for Go detects buggy and redundant code patterns. Like many language style guides, it provides a practical description of "do X instead of Y" for code. An example from the documentation explains:

```
You can use range on nil slices and maps, the loop will simply
never execute. This makes an additional nil check around the
loop unnecessary.
```

Regular expressions cannot generally recognize patterns like the `nil-check` simplification at the top of Figure 2.1. The traditional solution is to write a checker that visits the program's AST to recognize the pattern. Checker mechanics are hidden in an implementation, diverging from Figure 2.1's intuitive, syntactic before/after description. This programmatic implementation disconnects the checker code from the pattern, making it harder to understand and modify.

In our declarative approach, the user instead specifies a match-and-rewrite pattern that closely resembles and is as natural as the syntactic description. The *match template* for `nil-check` pattern is shown at the bottom of Figure 2.1. The syntax `:[`*identifier*`]` denote

[3] https://github.com/comby-tools/comby
[4] https://github.com/squaresLab/pldi-artifact-2019/blob/master/PullRequests.md
[5] https://staticcheck.io/docs/#overview

Before

```
if s != nil {
 for _, x := range s {
    ...
 }
}
```

After

```
for _, x := range s {
  ...
}
```

Match template

```
if :[var] != nil {
 for :[_] := range :[var] {
    :[body]
  }
}
```

Rewrite template

```
for :[_] := range :[var] {
  :[body]
}
```

Figure 2.1: *Top:* A textual description for simplifying a `nil` check Go code, taken from the Go `staticcheck` tool. *Bottom:* Our match template and rewrite templates for the `nil-check` pattern above.

holes with the name *identifier*. Every instance where the template matches source code produces an environment. An environment binds variables to string values (corresponding to syntax) in the source code. The identifier _ acts as an ordinary identifier where we don't particularly care to name the variable. Multiple occurrences of a variable in a template (like `var`) imply that matched values must be equal for matching to succeed.

The match template is expressive enough to match the syntactic structure of the general pattern in Figure 2.1. Three core ideas make this possible:

1. The *built-in* understanding that delimiters `{...}` in the match template denote well-balanced (and possibly nested) syntax that must correspond to delimited syntax in the source.

2. Any syntax besides balanced delimiters and holes in the template (e.g., non-whitespace tokens like `if`, `!=`, `range`) *correspond to concrete syntax* in the source. Whitespace in the template is interpreted as a regular expression matching one or more consecutive whitespace characters.

3. Match *rules* govern whether the match succeeds. Rules act as additional constraints on values in the environment (e.g., we check that matches of `:[var]` are equal).

Section 2.4 explains the matching approach in detail. After the matching phase, we can use a *rewrite template* to perform a change to the program. A rewrite template takes as input an environment, and substitutes variables for values. Our rewrite template for `nil-check` is also declarative (see Figure 2.1).

The match and rewrite templates are the only inputs needed to detect *and rewrite* instances of `nil-check`. Figure 2.2a is a real-world example where we applied our `nil-check` rewrite specification, producing the simplified code as-is in Figure 2.2b.[6] The rewrite is nontrivial: one match is contained inside another. Importantly, *each instance* matched by the template (which may be nested or occur in sequence) produces an environment to be instantiated using the rewrite template at that location. In this sense, our match-and-rewrite approach mimics traditional recursive, visitor-like tree traversals without a need for an actual AST visitor.

Indeed, presently, despite the fact that nothing about this approach is Go-specific, implementing the `nil-check` specification invariably requires creating or interfacing with a Go-specific tool operating on a parsed AST. One approach to multi-language support is an intermediate representation (e.g., `srcML` [150]), but this requires a translation layer and remains a programmatic approach. In language-specific approaches, tool design is dictated by the language's AST. This precludes, for example, the use of a general visitor framework for both Python and Go. This may be unavoidable due to language-specific features and tool application. However, for general syntactic transformations like Figure 2.1, our approach avoids repeated implementation effort across different languages. Moreover, interfacing with the AST is programmatic in language-specific AST visitor frameworks. Similarly, language workbenches supporting transformation for multiple languages (e.g., [55, 125]) generally define an AST representation per language with particular sensitivity for matching program terms. In this design, making transformation declaratively accessible requires an additional

[6]This code was merged into a popular Go repository: `https://github.com/sourcegraph/go-langserver/pull/324`. We did reindent the code using `gofmt` so that it conforms to stylistic conventions. Reformatting is not always necessary.

```go
func (c *SymbolCollector) addContainer(...) {
  if fields.List != nil {
    for _, field := range fields.List {
      if field.Names != nil {
        for _, fieldName := range field.Names {
          c.addSymbol(field, fieldName.Name)
        }
      }
    }
  }
  ...
}
```

(a) Highlighted lines 2 and 4 contain redundant `nil` checks in Go code: iterating over a container in a for loop implies it is non-nil.

```go
func (c *SymbolCollector) addContainer(...) {
    for _, field := range fields.List {
        for _, fieldName := range field.Names {
          c.addSymbol(field, fieldName.Name)
        }
    }
    ...
}
```

(b) Rewrite output simplifying the Go code above.

Figure 2.2: Redundant code pattern and simplification.

translation layer from a surface syntax mapping to terms (e.g., [201]). In general, the specificity of terms in AST definitions determines the granularity at which such terms can be matched using generic tree matching procedures. This specificity (which can result in definitions spanning hundreds of lines)[7] can increase the complexity of the surface syntax for complex transformations (e.g., when disambiguating syntax [119]). By contrast, we propose a particularly coarse representation for lightweight transformations that is loose with respect to matching and rewriting specific terms. Our declarative transformation approach notably skirts the issue of defining a concrete representation for manipulating code (like ASTs) or defining additional translation layers (e.g., [201]) by creating on-demand

[7]See, e.g., https://github.com/usethesource/rascal/blob/master/src/org/rascalmpl/library/lang/java/syntax/Java18.rsc

parsers that match syntax of interest. The foundation of this idea lies in first identifying a suitable grammar for expressing syntax of multiple languages, discussed next.

2.3 Dyck-Extended Languages

We use the Dyck language as an initial building block to form a general and extensible representation for syntactic rewriting. The Dyck language is simply an abstraction of well-nested expressions over a single pair of symbols that form an open and closing delimiter pair. I.e., taking the symbols "`(`" and "`)`" as delimiters, `()()` and `(())` are valid expressions in the Dyck language. A generalization of the Dyck language maintains a well-nested structure over n pairs of delimiting symbols, denoted D_n. Like other literature, we refer to D_n as "Dyck Languages". The definition of D_n is:

$$S \to \epsilon \mid SS \mid x_1\ S\ \overline{x}_1 \mid x_2\ S\ \overline{x}_2 \mid \cdots \mid x_n\ S\ \overline{x}_n$$

where ϵ represents the empty string, S is the only non-terminal symbol, and terminal symbols $(x_i, \overline{x}_i)$ for each $i \in n$ delimiter pairs. We denote the terminal alphabet for this grammar as $T = X \cup \overline{X}$ where $X = \{x_1, \ldots, x_n\}$ and $\overline{X} = \{\overline{x_1}, \ldots, \overline{x_n}\}$ are disjoint sets over opening and closing symbols, respectively.

Our key addition to the grammar allows interpolating strings between and inside balanced delimiters in D_n. We add the rule $S \to c$, for terminal c, where $c \in \Sigma$ for a finite alphabet Σ disjoint from $X \cup \overline{X}$. We call this a general Dyck-Extended Language (DEL) grammar. To be practically useful for parsing conventional programs, we specialize this grammar and impose:

1. a finite set of matching delimiters, e.g., `(`,`)`, `{`, `}`, `[`, `]` as found in common languages.
2. a refinement that strings produced from the finite set Σ is regular, where strings

correspond to tokens.

grammar ::= *term* `EOF`
term ::= `'('` *term* `')'` | `'{'` *term* `'}'` | `'['` *term* `']'` | *term term* | *token*
token ::= *whitespace* | *string_literal* | *comment* | *any_token*

---

whitespace ::= [`␣\n\t\r`]
string_literal ::= ⟨`"escaped string"`⟩
comment ::= ⟨single or multiline comment⟩
any_token ::= [...]+

Figure 2.3: A base grammar for parsing multi-language syntax. Productions above the line are common to many languages. Below the line, token structure may vary in minor ways (e.g., different string literal or comment syntax).

A DEL for multi-language transformation. We use the DEL definition shown in Figure 2.3 as a basis for conventional languages, which extends the grammar above. The extension defines concrete delimiters and partitions three kinds of tokens: whitespace, string literals, and comments. String literals can contain any character (with appropriate escaping, as usual). All other consecutive characters not already defined (i.e., ellipses denote any character excluding whitespace characters, string delimiters and comments) form an *any_token*. In practice, the finite character set of the language is that of typical source code (i.e., ASCII or Unicode character sets). The distinction between string literals and comments is significant because these are the primary categories that, if ignored, break the ability to recognize well-balanced delimiters in code. String literals and comments are simultaneously a source of syntactic variability and ambiguity across languages. Handling these categories explicitly across languages allows our approach to correctly identify nested terms within those languages.

There are two notable properties of the representation. First, a parser for this language preserves all syntax (including whitespace) and partitions all characters coarsely into one

of a few lexical constructs. This means that many separate lexical constructs in typical grammars are treated as one lexical construct under *any_token* (i.e., we don't distinguish keywords from variables). Second, hierarchical tree structure is determined solely by balanced delimiter syntax. Importantly, the representation preserves a well-formedness property of nested terms with respect to these delimiters when we rewrite code. Our intuition is that this representation models the essential concrete syntax for rewriting many conventional languages like C, Python, OCaml, etc., with expressivity to describe nested structures. At a high level, our representation can be seen as imposing a kind of s-expression representation on conventional language syntax, extending similar flexibility of lisp-like macro systems [45, 46, 86, 208] to many languages by accounting for varying syntactic idiosyncrasies (e.g., different sorts of balanced delimiter, comment, and string literal syntax) in a modular grammar.

In Figure 2.3, the delimited terms and token structure above the line denote commonly *shared* (though coarse) productions for many contemporary languages. Below the line, token syntax may *differ* across languages in minor ways (e.g., quotes used for string literals, raw string literals, or differing comment syntax). The crux of our representation is that it allows a modular and extensible parsing strategy. Using parser combinators [113], we design a modular skeleton parser for the shared constructs (above the line Figure 2.3) that is "open": parsers for token groups (below the line) plug into the skeleton parser. Our parser combinator architecture thus allows easy modification to handle syntactic idiosyncrasies at the token level, while preserving the essential CFL properties denoted by balanced delimiter syntax. In our experience, rewrite support for multiple languages with this architecture boiled down to tiny modifications in token parsers (typically one line of code) for handling individual language comment or string literal syntax. We now explain how the DEL representation, and the parse strategy for it, ties into matching and rewriting syntax.

2.4 The Rewrite Process

The rewrite process has two phases: (1) matching source code using a user-specified *match template* and (2) rewriting source code based on a user-specified *rewrite template*. We explain the intuition of our matching approach in Section 2.4.1. We introduce our key innovation in Section 2.4.2 that lifts the intuition of tree matching to a parser generator problem for modular grammars (as introduced in Section 2.3). We explain the rewrite phase in Section 2.4.4.

2.4.1 Declarative Matching: The Intuition

Suppose we have a DEL grammar parser for Go that produces DEL trees for Go code (i.e., the parser handles Go comments and string literals). The idea behind *declarative* match templates is that they translate directly to an underlying DEL parse tree to be matched against a DEL parse tree produced by source code. Match templates additionally contain holes that unify with syntax in the source parse tree. Successful matches produce environments binding variables to syntax, which are substituted into the rewrite template in the rewrite phase (Section 2.4.4)

Figure 2.4 visualizes a match of template to source for our motivating example. On the top is a parse tree of the source code; on the bottom, a parse tree for the template. Dashed lines indicate matched tokens. Concrete tokens match syntactically in template and source. Balanced delimiters (e.g., `{}` braces) form nodes corresponding to terms that contain nested child terms; when encountered in the template, balanced delimiters must match correspondingly in the source. Note that parse tree under in the simple DEL representation flattens traditional terms (such as the `if` condition and `for` assignment) if they are not enclosed by delimiters as specified for the language. Note also, however, that we *can* specify alphanumeric delimiters for a language. For example, we can specify that the delimiters `if`...`fi` should be well-nested for the Bash language.

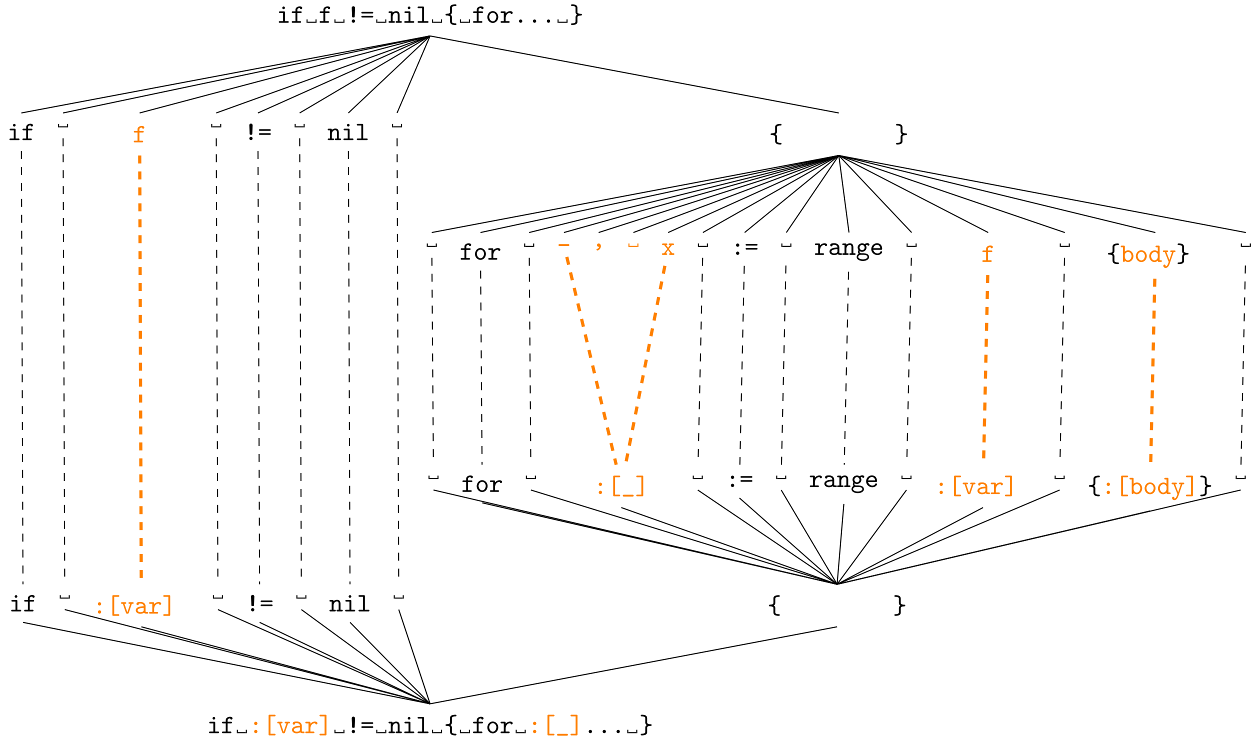

Figure 2.4: Visualization of matching for the `nil-check` pattern (Figure 2.1) using the DEL representation. On the top is a parse tree of source code; on the bottom a parse tree for the match template. Dashed lines represent matched terms. Thick lines indicate assignment of terms to variables.

Holes match to syntax based on (a) prefix and suffix matches of surrounding terms and (b) their level in the tree. As shown, the `:[_]` hole can only bind to sequences of terms inside the brace-delimited `if`-body. The matching semantics of `:[`*identifier*`]` is to match lazily up to its suffix. In practice, we do not need an explicit token class for the category of *any_token* in Figure 2.3: we simply treat these as individual characters (i.e., each character can be considered its own term). Thus, holes can bind to any sequence of characters (akin to regular string matching), restricted within its current level in the tree. A hole's suffix is terminated by (a) another hole or (b) the end of the sequence of characters (preserving proper nesting) on its level. For example, the hole `:[_]` matches “`_,␣x`” up to its suffix “`␣:=␣range␣`”, after which another hole occurs at the same level in the template. A hole immediately followed by another hole (e.g,. `:[1]:[2]`) implies that the first hole matches up to the empty string (i.e., `:[1]` will always be empty). We define one additional

metasyntactic form: `:[[`*identifier*`]]`, which matches one or more alphanumeric characters and `_`. Hole syntax is interpreted differently when enclosed by string delimiters for a given language. For example, it is possible to write `":[hole]"`, which will only match well-formed strings in languages that use double quotes. Comments are treated as whitespace; thus, comments can be captured by holes, but it not possible to explicitly match on comments currently.

We do not define additional metasyntactic operators (e.g., regex operators); *all* other syntax in the template relates to *concrete* matching.[8] The concrete specification of templates is separated from their *interpretation*, which must be implemented in the matching procedure. For example, delimiter syntax implies matching must descend into the tree. For convenience, we also interpret a single space in the template as matching repetitions of whitespace in the source by default.[9] This matching behavior can be modified to expect exact whitespace matching in the tool configuration, if desired.

Implementing the match procedure as described raises two intertwined but separate concerns: (1) producing parse trees from match templates and source and (2) matching templates to source code by traversing the trees. These present two design challenges: defining a concrete data structure to represent DELs (i.e., a tree, the output of parsing), and implementing a configurable tree traversal strategy to enable configurable and flexible match interpretations.

Our insight is that both challenges are solved simultaneously by lifting tree construction and matching to a modular parsing problem. Doing so enables structural matching relating template and source (a) without any concrete representation at all and (b) with configurable matching represented directly by parsers, allowing for syntactic differences across languages, and the ability to match character sequences (e.g., like regular string matching).

---

[8]Nothing stops us from complicating templates with more metasyntax; in practice we find doing so eventually limits declarative specification.

[9]This is desirable in our running example when we don't particularly care about the indentation size or other additional whitespace.

2.4.2 Parser Parser Combinators for Matching

The key innovation behind our approach is an extensible procedure via parser combinators [113] for structural matching that relates match templates and source code. The core idea of parser combinators is to model parsers as functions that can be composed using higher-order functions (combinators) to implement grammar constructions. Higher-order functions make the parsing process separable. Thus, composable parser combinators enable a modular parsing procedure for the modular grammar we developed in Section 2.3. We might, for example, use a modular parsing strategy to produce DEL parse trees for matching. However, this still leaves open the problems of (a) mapping parse results to a tree definition, then (b) defining a configurable match traversal strategy over this representation, and (c) creating an accessible declarative language to mapping user-specified templates to terms in the tree definition (as in [201]).

Instead, we use parser combinators in a new way. We first recognize that parser combinators can be composed to parse DEL languages. Such a DEL parser, when running on source code, performs the analogous role of matching valid parses of concrete syntax (to generate, in this case, a parse tree). Interestingly, a declarative match specification is nothing other than a description of particular concrete syntax (interpreted by a DEL parser), combined with holes, that is to be matched. Thus, a match specification can be interpreted as a description of a parser for matching, rather than a concrete tree to unify against a parse tree of source code. In particular, it describes a valid parse for a DEL language fragment.

This insight leads to a solution where our approach generates on-demand parsers for DEL language fragments from match templates. We develop what we call a Parser Parser Combinator (PPC): a parser-generating parser using parser combinators. The approach works by implementing a parser for match templates *whose output is a parser* (constructed from parser combinators at runtime) that matches syntactic patterns according to the

template. The match template PPC behaves as follows:

- When parsing a DEL token in the template (e.g., a string like `if`), generate a parser for that token.
- When parsing whitespace, generate a parser to consume whitespace.
- When parsing nested syntax (indicated by delimiters), generate a parser for nested syntax between those same delimiters.
- When parsing holes (`:[...]`), generate a parser that binds syntax to the identifier and continues lazily up to the suffix in the template (i.e., generate a parser implementing the match semantics described in Section 2.4.1 and store the result in the parser state).

The PPC automatically chains generated parsers in these categories *as it parses* the template at runtime. The resultant generated parser is lazy (i.e., it is a partial function) which encodes exactly the behavior for matching syntax of interest. The appeal of this approach is that we can now simply run the resultant parser directly on source code. The result of the generated parser is only the environment binding variables to syntax for successful matches. No intermediate data structure is needed to perform any matching.

The PPC approach also offers key advantages for configuring matching. First, the *interpretation* of match template syntax (e.g., match semantics of holes, or whether to treat whitespace significantly) is easily modified by augmenting the parser generating behavior of the PPC when it encounters any particular kind of syntax in the template. Second, our approach allows matching substrings familiar in regular expression-based code changes (e.g., [27]) as opposed to restricting matching to complete tokens or terms. In practice, match template content for non-whitespace characters maps to a matching parser character-for-character (analogous to lexing, but without classifying specific tokens). Importantly, well-formedness of nested delimiters is preserved when matching character sequences—a character cannot match, for example, an unbalanced parenthesis character unless it occurs

inside a string literal or comment. Third, the PPC is extensible: it is a module exposing hooks in the parser-generation routine (which is modular, like the DEL grammar) so that parsing language-specific syntax (e.g., strings, comments) is expressed in the generated parser. The PPC is implemented as an OCaml functor that accepts a module with the following signature:

```
module type Signature = sig
  val user_defined_delimiters : (string * string) list
  val string_literals : string
  val comments : comment list
end
```

The PPC is extended using small definitions corresponding to this signature, which express language-specific syntax properties. For example, the three definitions below form a `Base.Syntax` module defining the syntax of the base DEL parser:

```
let user_defined_delimiters = [ "(", ")"; "{", "}"; "[", "]"]
let string_literals = []
let comments = []
```

The C parser extends the base parser. It includes the base delimiter syntax, and adds the syntax for string literal and comment syntax using the prescribed definition names:

```
include Base.Syntax
let string_literals = ["\""; "'"]
let comments = [ Multiline ("/*", "*/"); Until_newline "//" ]
```

Constructors like `Multiline` denote, e.g., that C-style block comment syntax should be used to parse multiline comments.[10] Parsers are created from these syntax definitions and embed into the PPC to parameterize *both* the significance of custom syntax in user-supplied match templates *and* the significance of the same custom syntax when the resulting matcher runs on source code. For example, the C definitions above reflect in the user-specified

[10]For brevity we elide other optional definitions that can further refine the PPC, e.g., raw string literals that have different escaping criterion.

template (the parser now understands that holes inside string delimiters should match within string delimiters) and ensures that parentheses inside strings in the target source code (like `"("`) do not affect the parsing of balanced parentheses.

We implement our PPC using a left-to-right, top-down parsing strategy. Any ambiguity in the grammar is resolved by the ordered choice combinator (as in, e.g., [90, 137]), where terms are parsed similarly to the order of Figure 2.3.[11] The PPC implementation is roughly 500 lines of OCaml code. In practice, applying our PPC to 12 languages requires fewer than 12 lines of additional code to account for the syntactic differences in comments and string literals. Command-line flags activate parsers for a particular language.

Match contexts. Figure 2.4 illustrates matching one instance of a template to an exact match of source code. Similarly, PPCs produce a parser that match one instance specified by the template. In practice, we typically want to find multiple matching instances in source code. We call each matching instance a *match context*. A match context describes (1) the range of characters in the source code matching the template in its entirety and (2) the unique environment binding holes to values for that instance.

For practical reasons, our objective is not to find *all* matching contexts, but rather non-overlapping matches over sequences. For example, using the match template `a␣b␣a` and a source `a␣b␣a␣b␣a` produces only the match `a␣b␣a␣b␣a` and not `a␣b␣a␣b␣a`. Disallowing sequentially overlapping match contexts allows an unambiguous rewrite result. We produce match contexts by shifting over the source code left-to-right and attempting to parse at each point. If the PPC-generated parser succeeds at any point, we record the match context (i.e, the match range and environment). We then shift to the end of the match range and continue trying to parse at each point until the end of source is reached. Our tool allows the same procedure to be applied successively on content in match environments, thereby recursively rewriting nested matches.[12]

[11] We overlook the possibility of left-recursion in this grammar, which we gave to simplify presentation.
[12] Alternatively, the pass may be rerun to a fixpoint on the entire file.

2.4.3 Match Rules

Match rules apply additional constraints on matching. Match rules are high level predicates on environments in match contexts. Our motivating example demonstrates the utility of predicates on matching. For example, we can check that the variable ranged over is the same as the variable compared to `nil`. As in Figure 2.1, using the same variable identifier for multiple holes adds the implicit constraint that values matched by these holes should be equal. This constraint is really syntactic sugar for a larger feature set of constraints that can be specified explicitly and declaratively in a small DSL-like notation.

A match rule may accompany a match template and has the form `where <...>` and a list of comma-separated expressions and evaluates to a boolean value. Commas are interpreted as logical conjunction over the expression list. If the value is true, a match succeeds. The default match rule is `where true`. The grammar is:

```
grammar ::= "where" ⟨expression⟩ ⟨"," expression⟩*
expression ::=
    | "true"
    | "false"
    | ⟨atom⟩ "==" ⟨atom⟩
    | ⟨atom⟩ "!=" ⟨atom⟩
    | "match" ⟨atom⟩ "with" ⟨branch⟩ ⟨"|" branch⟩*
atom ::= ":[" ⟨variable⟩ "]" | ⟨ "string literal" ⟩
branch ::= ⟨atom⟩ "->" ⟨expression⟩
```

Figure 2.5: A simple constraint grammar for match rules.

Operators `==` and `!=` check for syntactic equality (resp. inequality) on atoms. There are only two atoms: one for variables (expressed in hole syntax) and string literals. The `match` expression applies (disjunct) conditional constraints on values bound to holes.

It is sometimes useful to treat values as templates in `match` expressions, rather than string values. When string literals contain hole syntax, they are interpreted as match

templates. In this case, we may write *match templates* for the antecedent in match cases which produce match contexts when evaluating the branch expression. For example,

```
where match :[expr] with | ":[1] < :[2]" -> :[1] == :[2]
```

evaluates `:[1] == :[2]` *if* the value bound to `:[expr]` produces a match context for the match template `:[1] < :[2]`. The branch expression must evaluate to true for all match contexts produced by the template.

We do not give extensive examples of transformations with rules here (the refactor patterns we pursue in this chapter do not generally require constraints beyond checking syntactic equality). More complex rules play a significant role in Chapter 3.

2.4.4 The Rewrite Phase

The rewrite phase takes all match contexts produced by the match phase and substitutes the environment variables for values in a user-specified rewrite template. The rewrite template can be seen as a partial function. We say that a rewrite template is *instantiated* by an environment when all variables in the rewrite template are substituted for values. After a rewrite template is instantiated, it replaces, in-situ, the entirety of matched content for a given match context. Thus, there are two substitution passes: (1) substitution to instantiate rewrite templates and (2) contextual substitution of instantiated templates in the source. The rewrite procedure applies recursively for nested matching contexts, and are substituted bottom-up as in the motivating example.

Well-balanced output. During the match phase, only well-balanced terms can be matched by holes. By extension, rewrite output is well-balanced if and only if the rewrite template is well-balanced. Thus, rewrites preserve key syntactic properties of well-balanced delimiters found in most languages.

Rewriting comments. Associating comments across program manipulation is a known

and long-standing problem, and not specific to our approach [156]. Holes *do* capture comments and can be recovered during rewriting. However, comments are ignored when interpreting match templates (otherwise, templates would require the user to anticipate comment location and content to match other significant syntax). Matched syntax is replaced according to the rewrite template as usual, and when matching on comments is not significant, those comments may be erased by the rewrite if they are not captured by holes.

The underlying problem is that we cannot anticipate how comments may be associated with syntax for arbitrary languages. In lieu of better language design (which appears inapplicable to contemporary languages) we note that inference techniques or explicit comment support (e.g., adding an extra dimension of comment contexts) could provide solutions for manipulating comments.

2.5 Evaluation

We evaluate `Comby`, an implementation of our rewrite process (Section 2.4) for DELs (Section 2.3) using Parser Parser Combinators (Section 2.4.2). Section 2.5.1 evaluates `Comby` on 12 languages comprising 185 million lines of code using 30 transformations. We demonstrate that our approach is fast; enables simple, declarative specifications for nontrivial transformations; and that it is effective in *real* (not just realistic) settings, to date producing over 50 changes merged into 40+ of the most popular open source projects across 10 languages.

In Section 2.5.2 we compare our approach to nine existing language-specific checker and rewrite tools. We show that our multi-language approach produces accurate syntactic checking and transformation for many syntactic properties covered by current tools, while requiring far less implementation effort. Our results indicate that `Comby` is efficient, and meets the demands of real-time development [84, 166]: transformation response time is in the 100-400ms response time per file. Section 2.5.3 provides further discussion and

Table 2.1: We ran 1–3 rewrite patterns (**Pattern**) per language (**Lang**) on the top 100 most-starred GitHub projects for that language. **K Files** and **MLOC** is the aggregate thousands of files and millions of lines of code, respectively. **Proj** is the number of projects where patterns match; **#M** the number of matches; **Time**, the aggregate time to match/rewrite over all files.

Lang	**K Files**	**MLOC**	**Pattern**	**Proj**	**#M**	**Time**
Go	131.4	48.8	nil-check	40	372	7m56s
			str-contains	16	29	3m02s
Dart	28.2	4.9	slow-length	6	35	24s
			where-type	10	165	35s
Julia	5.2	0.9	simple-map	5	8	23s
			tweaks (3)	12	38	45s
			redun-bool	1	1	24s
JS	66.1	9.2	redun-bool (2)	3	89	7m14s
Rust	22.0	3.8	short-field	42	305	21s
			redun-pat (4)	2	12	1m15s
Scala	41.9	5.2	parens-guard	10	43	30s
			forall	2	5	23s
			count	13	46	23s
Elm	4.8	0.9	dot-access	2	3	13s
			pipe-left	4	10	13s
OCaml	25.6	6.3	include-mod	11	77	2m01s
			include-func	4	5	2m04s
C/C++	166.6	85.5	no-continue	2	5	5m58s
Clojure	5.1	0.7	simpl-check (4)	34	330	21s
Erlang	13.2	4.3	infix-append	25	187	25s
Python	34.5	15.2	dup-if-elif†	4	10	6m59s

limitations.

2.5.1 Real, Large Scale Multi-Language Rewriting

Experimental setup. We cloned the top 100 most popular repositories on GitHub (ranked by stars)[13] for each of 12 languages. We balanced imperative and functional languages (some mature, others more recent) to demonstrate broad applicability. The 1,200 repositories comprise over 185 million lines of code. We wrote 1–3 rewrite patterns per language (we elaborate below). We ran each rewrite pattern in parallel by file across 20 cores (Xeon E-2699 CPU, Ubuntu 16.04 LTS server, limiting experiments to 1GB of RAM). Matching was set to timeout after 2 seconds per file. Roughly 2% of Python files timed out (large, autogenerated code), while timeout for files across other languages accounted for less than 0.5%.

Choosing rewrite patterns. We chose patterns from existing linters, checkers, rewrite tools, and style guides that were likely to improve code readability or performance. Since we submitted many of our transformations to active code bases, we avoided stylistic patterns that were a matter of taste. The selected patterns either (1) remove clearly redundant code; (2) simplify readability; (3) replace functions with more performant versions; or (4) identify clearly buggy duplicate checks. We include nontrivial patterns that match on balanced delimiters for every language.

This part of our evaluation prioritizes (a) breadth of language application and (b) improving active real code. We therefore applied 1–3 choice patterns per language rather than a comprehensive catalogue of patterns for any particular one.

Results overview. Table 2.1 shows the results of running multiple rewrite patterns across 12 languages. The rewrite patterns are shown in the **Match Template** and **Rewrite Template** columns of Figure 2.6. As written, these are the actual inputs to `Comby` (no extra

[13] In October, 2018.

metasyntax or programming required).[14] We applied 30 unique patterns to our data set. Some patterns, like "tweaks" and "redun-bool" check for similar patterns with syntactic variations (we show a representative transformation for these in the table). The number of variations is indicated in parentheses in the **Pattern** column, and the running time is an aggregate over variations. In aggregate, running all patterns takes about 42 minutes and parses 282 million lines of code (by common word count including newlines and comments).

Rewriting is fast: 13 out of 21 patterns terminate in under a minute for all 100 projects of that language, while the longest runtime for a pattern takes roughly 8 minutes (`nil-check` for Go code). Generally, longer runtimes correspond to larger codebases of popular languages (e.g., Go, Javascript, Python, and C). Match templates and source code content also affect runtime: The Go `nil-check` pattern takes more than twice as long compared to the `str-contains` pattern due to increased cases of matching the `if :[1]...` pattern.

The number of projects for which a rewrite pattern applies range from 1 to 42 (**Proj**) and the number of matches from 1 to 372 (#**M**). To demonstrate our multi-language rewrite approach in a real setting, we submitted a subset of these changes for merging into upstream repositories via GitHub pull requests (PR). This demonstrates both the opportunities for syntactic transformation in large, highly popular repositories and the ability of our approach to produce actionable, correct, and automatic syntactic changes in this setting. Submitting all changes was not appropriate because changes may touch code dependencies not part of the main project, or code that explicitly tests for bad patterns (e.g., linter tests). Unless stated otherwise, we used the following criteria to submit pull requests: (1) the project must be active (code had been committed in the previous 30 days); (2) the changes should not affect files under a `test` or `vendor` directory; (3) we prefer projects where we can validate syntactic changes via a continuous integration build.

The results of our PR campaign are shown in Table 2.2. We submitted PRs to 50 projects. The **Merged** column shows the number of syntactic changes submitted and the

[14] Modulo newline stripping, for presentation.

<table>
<tr><th>Lang</th><th>Pattern</th><th>Match Template</th><th>Rewrite Template</th></tr>
<tr><td>Go</td><td>nil-check</td><td><code>if :[1] = nil { for :[2] := range :[3] {:[4]} }</code></td><td><code>for :[2] := range :[3] {:[4]}</code></td></tr>
<tr><td></td><td>str-contains</td><td><code>if strings.Index(:[1], :[2]) != -1 {:[3]}</code></td><td><code>if strings.Contains(:[1], :[2]) {:[3]}</code></td></tr>
<tr><td>Dart</td><td>slow-length</td><td><code>if (:[1]s.length = 0)</code></td><td><code>if (:[1]s.isNotEmpty)</code></td></tr>
<tr><td></td><td>where-type</td><td><code>.where((:[1]) => :[1] is :[[type]])</code></td><td><code>.whereType<:[type]>()</code></td></tr>
<tr><td>Julia</td><td>simple-map</td><td><code>map(:[1]->:[[x]](:[1]), :[y])</code></td><td><code>map(:[x], :[y])</code></td></tr>
<tr><td></td><td>tweaks (3)</td><td><code>ceil(:[1]/:[2])</code></td><td><code>cld(:[1],:[2])</code></td></tr>
<tr><td></td><td>redun-bool</td><td><code>(:[1] || :[1])</code></td><td><code>(:[1]) # redundant expr</code></td></tr>
<tr><td>JS</td><td>redun-bool (2)</td><td><code>(:[1] && :[1])</code></td><td><code>(:[1]) // redundant expr</code></td></tr>
<tr><td>Rust</td><td>short-field</td><td><code>{ :[[field]]: :[[field]] }</code></td><td><code>{ :[field] }</code></td></tr>
<tr><td></td><td>redun-pat (4)</td><td><code>if let Ok(_) = Ok:::[t](:[v])</code></td><td><code>if Ok:::[t](:[v]).is_ok()</code></td></tr>
<tr><td>Scala</td><td>parens-guard</td><td><code>for {:[body]if (:[1]) }</code></td><td><code>for {:[body]if :[1] }</code></td></tr>
<tr><td></td><td>forall</td><td><code>.foldLeft(true)(:[1] && :[1])</code></td><td><code>.forall(identity)</code></td></tr>
<tr><td></td><td>count</td><td><code>.filter(:[1]).size</code></td><td><code>.count(:[1])</code></td></tr>
<tr><td>Elm</td><td>dot-access</td><td><code>List.map (\:[[x]] -> :[[x]].:[[f]]) :[[vs]]</code></td><td><code>List.map .:[f] :[vs]</code></td></tr>
<tr><td></td><td>pipe-left</td><td><code>|> List.map :[[fn1]] |> List.map :[[fn2]]</code></td><td><code>|> List.map (:[fn1] >> :[fn2])</code></td></tr>
<tr><td>OCaml</td><td>include-mod</td><td><code>module :[m] = struct include :[[x]] end</code></td><td><code>module :[m] = :[x]</code></td></tr>
<tr><td></td><td>include-func</td><td><code>module :[m] = struct include :[[x]](:[b]) end</code></td><td><code>module :[m] = :[x](:[b])</code></td></tr>
<tr><td>C/C++</td><td>no-continue</td><td><code>for (:[1]) { continue; }</code></td><td><code>for (:[1]) {}</code></td></tr>
<tr><td>Clojure</td><td>simpl-check (4)</td><td><code>(= :[1] nil)</code></td><td><code>(nil? :[1])</code></td></tr>
<tr><td>Erlang</td><td>infix-append</td><td><code>lists:append(:[1], :[2])</code></td><td><code>:[1] ++ :[2]</code></td></tr>
<tr><td>Python</td><td>dup-if-elif†</td><td><code>if :[1]::[_] elif :[1]:</code></td><td><code>if :[1]::[_] # duplicate check</code></td></tr>
</table>

Figure 2.6: The match-rewrite templates for the patterns in our large-scale evaluation. We performed additional preprocessing for the `dup-if-elif` Python pattern (†).

Table 2.2: Pull Request Results. **Merged** shows the number of changes submitted, and outcome: ✓means changes were merged; ⏲ means a PR is in progress; – means changes were rejected for reasons unrelated to their content; ✗ indicates incorrect changes. †: the `redun-bool` and `dup-if-elif` required manual fixes.

Lang	Pattern	Project	#KLOC	#M	Merged
Go	nil-check	go-langserver	247.5	2	✓ 2
		helm	50.2	1	✓ 1
		prometheus	902.4	2	✓ 1
		rclone	596.9	10	✓ 1
		go	1,703.2	3	✗ 2 ✓ 1
		cli	10.4	1	✓ 1
		rancher	2,695.2	14	✓ 1
		kubernetes	4,236.6	22	⏲10
	str-contains	gorm	14.6	2	✓ 2
		vault	1,553.1	1	✓ 1
		vitess	297.7	2	✓ 2
Dart	slow-length	sdk	2,361.3	18	✓ 3
		sqflite	6.9	1	✓ 1
		over_react	23.3	1	✓ 1
Julia	simple-map	Dagger.jl	3.7	1	✓ 1
		JLD.jl	4.5	1	✓ 1
		IJulia.jl	2.7	1	✓ 1
	redun-bool†	DataFramesMeta.jl	1.5	1	✓ 1
JS	redun-bool†	Rocket.Chat	149.2	3	✓ 3
		angular	297.8	1	✓ 1
Rust	short-field	rust	1,004.8	133	✓68
		clap	33.6	6	– 6
		conrod	34.1	5	⏲ 5
		coreutils	45.5	1	✓ 1
		exa	9.7	1	✓ 1
		fd	3.9	1	✓ 1
		gluon	66.4	6	✓ 6
		hematite	3.9	2	✓ 2
⋮	⋮	⋮	⋮	⋮	⋮
Scala	forall	scala-js	190.6	1	✓ 1
		shapeless	47.1	4	– 4
	parens-guard	Ammonite	22.5	1	✓ 1
	count	FiloDB	44.9	7	✓ 7
Elm	pipe-left	elm-graphql	36.4	1	✓ 1
		node-test-runner	40.1	2	– 1
OCaml	include-func	bap	100.7	1	✓ 1
		pyre-check	79.4	1	✓ 1
		hhvm	213.5	1	✓ 1
	include-mod	base	43.1	1	✓ 1
		opium	1.6	1	✓ 1
		flow	147.8	2	✓ 2
		owl	131.0	12	✓12
C/C++	no-continue	radare2	745.3	4	✓ 4
		php-src	1,428.4	1	✓ 1
Clojure	simple-check	pedestal	18.7	12	✓ 1
Erlang	infix-append	zotonic	102.2	5	✓ 5
		lasp	15.7	4	✓ 4
		kazoo	345.8	1	⏲ 1
Python	dup-if-elif†	matplotlib	229.3	1	✓ 1
		zulip	140.5	1	– 1
		powerline	30.0	1	✓ 1

outcome:

- ✓: 43 projects accepted more than 50 individual changes.
- ◷: 3 projects have a PR in progress, or received no response.[15]
- –: 4 projects rejected changes unrelated to change content.
- ✗: Only two changes were outright wrong.

The number of PR changes in the **Merged** column are either fewer or equal to the number of **Matches**, since we filtered out matches in test or vendor directories. Across all projects, the maximum time for a single refactor pattern is 19 seconds (the nil-check pattern for `Kubernetes`, which is the largest project in our data set). The median time per pattern, per project is 0.55 seconds. We elaborate on patterns and results for each language below.

Go. The `nil-check` pattern identifies redundant `nil` checks, as in our motivating example. The `str-contains` rewrite uses a clearer `string.Contains(...)` call compared to checking string indices for substring containment (see Fig 2.6). Changes from all of our PRs are accepted except for one, which went stale. Our PR to the `Go` compiler highlights an interesting case in that it is the only case where a syntactic change was outright wrong. The problem is that removing the check on `:[1]` in the pattern is only allowed on maps and slices, but not pointer types. In the Go PR, two cases compare to pointers. In the remaining PRs, these comparisons were done against appropriate types, and safe to remove. We make two observations. First, purely syntactic transformations are vulnerable to language design where equivalent syntactic changes have different semantic implications. Thus, while our approach suffers from syntactic ambiguity, we note that this ambiguity is a shortcoming that should be, in principle, avoided during language design for purposes of program manipulation [156]. Second, type information aids legal and complex syntactic changes (e.g., in refactorings [187]) and complements our approach. We note that recent uptake

[15] In April, 2019.

in the Language Server Protocol (LSP)[16] presents an elegant method for incorporating type information into our approach. With LSP, we can use external language servers (e.g., the `go-langserver`) to lookup types without having to parse, maintain, or persist type information for rewriting.

Dart. The `slow-length` pattern improves performance when checking the length of iterable containers (e.g., lists). The `where-type` pattern improves readability and removes redundant runtime checks. Both patterns are in the Effective Dart usage guide [28]. Because the `sdk` project does not have a CI build on GitHub, we validated a subset of syntactic changes locally. The `slow-length` pattern is susceptible to syntactic ambiguity: if the object is a string, there is no corresponding `isNotEmpty` method. As a heuristic, we changed the pattern to match variables ending in 's' as a likely indication that an Iterable object was checked (e.g., matches included '`columns`', or '`argNames`'). This presents an interesting case where matching at the character granularity can be desirable. The changes using this heuristic were all successfully validated by CI builds or local compilation. Projects with matches for `where-type` were all inactive or deprecated, except for the Dart `sdk`. Not all objects implement the `whereType` method, so the rewrite may fail to compile (none of the changes we validated failed to compile). In both cases, additional type or interface information complements our approach.

Julia and Javascript. The `simple-map` pattern removes redundant code produced by anonymous function syntax. The `tweaks` patterns can improve performance for floating point operations in tight loops. Both of these patterns are taken from Julia's style guide and performance tips. We discovered 38 cases where `tweaks` could potentially improve performance (Table 2.1), but this required benchmarking individual project performance before and after the change. Due to the extra burden of validating performance effects, we skipped PRs for this pattern.

[16] https://microsoft.github.io/language-server-protocol

We also checked Julia and Javascript code for redundant boolean expressions. On the one hand, this check is desirable because the pattern clearly indicates redundant or (more often) wrong code. On the other, it can be hard to predict a correct rewrite output when the code is wrong. By default, our rewrite output produces a semantically equivalent change with a comment to note the problem. Overall, this pattern demonstrates fast matching capability with optional rewriting. The one issue in Julia code and three issues in Javascript needed manual fixing;[17] in the case of `angular`, the expression was redundant and corresponds to the rewrite template `:[1]`.

Rust. The `short-field` pattern removes redundant single named fields for data structures (note we use `:[[..]]` match notation to limit matching to single identifiers). The `redun-pat` eliminates redundant `let...` pattern matching for option and result types. Both checks are implemented in an existing rust linter. Our largest pull request updates 68 instances where the Rust compiler now uses shorthand notation for single field structures (additional matches exist inside test directories). The PR for `clap` was closed because the code is being removed. We did not find real cases of `redun-pat`.

Scala. All three Scala patterns improve readability, and are implemented in existing tools (`scalafmt` and `scapegoat`). The PR for `shapeless` was rejected because the original syntax explicitly tested `foldLeft` behavior. This was one of our initial submissions, and influenced our decision to ignore files in test directories. Three eligible changes for `no-guard-paren` broke whitespace indentation and required reformatting; we skipped these and submitted a single PR where indentation was preserved. We discuss reformatting more generally in Section 2.5.3.

Elm. The Elm patterns simplify syntax by removing redundant anonymous functions (`dot-access`) and composing sequential map operations over list elements (`pipe-left`). We focused on `List` patterns, being a common data structure. Variants for these patterns can

[17]e.g., `(isnan(a[i]) || isnan(a[i]))` became `(isnan(a[i]) || isnan(b[i]))`

substitute `List` for `Array` or `Set`, but these are less common in practice. The `dot-access` pattern only rewrote code for two inactive projects (Table 2.1), so we did not submit pull requests for these. One `pipe-left` change was accepted, while the other was rejected because it touched files part of external dependencies.

OCaml. The `include-mod` and `include-func` removes redundant module include syntax for modules and functors respectively. The majority of `include-mod` matches are in compiler tests, which we did not submit in PRs. In some cases, the `include-func` pattern required reindentation because the module body `(:[b])` captures syntax that may span multiple lines. All of these PRs were merged.

C/C++. The `no-continue` pattern removes redundant continue statements inside `for` loops. The pattern is based on a check in `clang-tidy`. A number of variants are possible (e.g., replacing `for` with `while` or `do-while` syntax). We found two real cases of redundant `continue`s in the C/C++ dataset. Both of these PRs have been accepted.

Clojure. The `simpl-check` pattern for Clojure simplifies conditional checks. Syntactic variants include testing boolean values with `true?` and `false?` checks. All of the matches in the Clojure data set (Table 2.1) occurred in inactive projects, except for one. That PR was successfully merged.

Erlang. The `infix-append` pattern is part of the Erlang syntax tools suite. It replaces a `lists:append` function call with a terse `++` infix operator. Two PRs were accepted, while one remains open.

Python. The `dup-if-elif` pattern detects identical conditional checks on both branches of an `if-elif`. Matching on possibly nested and indentation-sensitive `if` statements is challenging compared to concrete syntax (e.g., braces) because whitespace is typically treated insensitively in many languages. We can implement an additional parser combinator hook to support layout-sensitive parsing in the PPC, but currently `Comby` treats indentation insensitively. However, our rewrite approach presents a convenient and interesting alternative

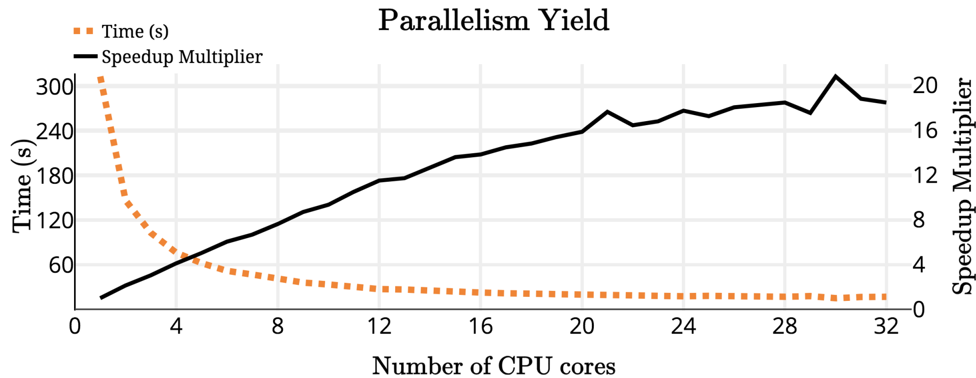

Figure 2.7: Parallelism yield running the `nil-check` pattern on Kubernetes (4.2 MLOC) up to 32 cores. Parallelism scales linearly up to 12 cores.

via syntax preprocessing. We used Python's `pindent`[18] utility to annotate the end of indentation-sensitive blocks with comments. We then ran a rewrite rule to convert block annotations to braces compatible with existing delimiter parsers. The preprocess step took approximately 6 minutes and performed 600K rewrites on the Python data set.[19] We discovered three legitimate `dup-if-elif` matches. Like the `redun-bool` pattern, the pattern identifies clearly buggy code and required manual fixes for the two merged PRs. In the remaining PR, we were asked to perform a larger refactoring that removes references to the buggy function so that it can be removed entirely.

Parallel performance and memory use. To quantify memory use and parallelism performance for large projects, we benchmarked our approach on Kubernetes (4.2 MLOC, the largest project in our PR campaign) with the `nil-check` pattern. We ran a precise memory profile of our implementation using OCaml Spacetime,[20] on a single core. Maximum memory use was only 44 MB while the median was approximately 10 MB. Low memory use is attributed to the PPC generating a lazy parser that persists only matching syntax per file.

We evaluated speedup of our implementation up to 32 cores. Figure 2.7 summarizes our

[18] https://github.com/python/cpython/blob/master/Tools/scripts/pindent.py
[19] The `pindent` preprocess step is also invertible.
[20] https://caml.inria.fr/pub/docs/manual-ocaml/spacetime.html

result. Performance scales linearly up to 12 cores before tapering off. Performance deviates slightly for higher numbers of cores due to heuristic partitioning of work (i.e., files) per core.

2.5.2 Comparison with Existing Tools

Experimental setup. We compare `Comby` to nine language-specific checker and rewriting tools in terms of (1) syntax checking and rewrite accuracy, (2) implementation burden, and (3) runtime. The tools provide a variety of functions including refactoring, code simplification, and linter checks. Four tools only check syntax, and do not rewrite. We targeted real-world projects where possible; otherwise, we applied patterns on the tool's test suite. We opted to use tests when isolating transformation behavior in tools was difficult. For example, some tools implement multiple syntactic checks under a single command line flag. Individual tool configuration also hinders a direct speed comparison. We provide the wall-clock times for running each pattern to indicate that our running time is competitive, but do not generally claim to be faster.

Experiments were run on a single core (2.2 GHz Core i7) with 1GB RAM. The hardware is comparable to a modern single-user developer machine, and supports our claim that our approach meets demands of real-time development response times.

We prioritized demonstrating breadth of application across languages, and include only a selection of patterns for eight of the nine languages (many more patterns are possible). For one functional language (Clojure) we implemented multiple patterns to evaluate how comprehensively our approach can express syntactic transformations compared to a production rewrite tool.

Tool comparison. Table 2.3 shows results: `Comby` produces identical or similar functionality for detecting and rewriting the investigated patterns. The `2to3` tool (part of `cpython`) assists in refactoring Python 2 code to Python 3. `Comby`'s output is identical to `2to3`'s. `Comby` also produces the same output for two patterns implemented by `Scalafmt`, a Scala formatter

Table 2.3: Tool Comparison. We compare against 9 existing checker and rewrite tools for multiple **Patterns**. **(R)** in the **Tool** column means that the tool offers rewrite capabilities; remaining tools only emit warnings. **Target Kind** describes the transformation target: either a selection of code extracted from a popular open source project, or tests that are part of the tool. **Impl. (LOC)** refers to the implementation's number of lines of code. In general, the implementation burden for rewriting in our approach (**Us, Impl.**) is less compared to that of language-specific implementations for the set of syntactic properties considered. The exception is `kibit`, which is comparably terse due to a logic programming representation for rewrite rules.

Lang	**Tool**	**Pattern**	**LOC**	**Matches**	**Time (ms)**		**Impl. (LOC)**		**Target Kind**
					Tool	**Us**	**Tool**	**Us**	
Python	2to3 **(R)**	next	6,908	11	911	486	76	4	osf.io
		range	13,057	14	1,309	895	53	8	
Erlang	erl-tidy **(R)**	append	4,891	2	1,426	277	8	2	ejabberd
		map	11,389	16	1,633	389	20	2	
Elm	elm-lint	map	64	1	1,712	124	67	8	graphql
		dot-access	153	2	2,727	130	51	2	take-home
Go	gosimple	nil-check	236	1	1,315	130	39	5	helm
		str-contains	467	1	2,409	132	65	2	gitea
Dart	linter	where-type	43	3	1,718	122	74	9	tests
		slow-length	518	1	1,808	134	89	2	sqflite
Rust	clippy	redun-pat	71	4	115	137	74	11	tests
		short-field	74	5	148	141	38	5	
Scala	Scalafmt **(R)**	curly-fors	32	2	1,312	154	80	6	tests
		parens-guard	41	2	1,451	270	32	2	
C++	clang-tidy **(R)**	no-continue	2,143	1	325	155	15	2	radare2
Clojure	kibit **(R)**	various (81)	182	97	6,840	164	215	240	tests

rewrite support tool. `Comby`'s output is identical to `erl-tidy`, an Erlang refactoring tool, except for one match case of the `map` pattern. The output differs because `erl-tidy` creates unneeded fresh named variables in the output. `Comby` detects the same syntax issues found by a subset of linter checks implemented for Go, Dart, Elm, and Rust across respective projects and linter tests.

Implementation effort and size. We manually reviewed code in existing tools to determine the implementation size (**Impl.**), a proxy for implementation effort. Checks were generally implemented at the function level, or within a larger function. We manually isolated check functionality within these functions, stripped newlines, and counted the lines of code. Generally, the implementation burden is far lower with our declarative match and rewrite templates.

Speed. All `Comby` transformations complete in under 400ms per file (Python's `next` and `range` patterns run on multiple files); thus, syntactic transformation is responsive enough to integrate with real-time development.

Parsing robustness. Because existing tools require ASTs, some do not work if the target file cannot be correctly parsed. For example, `clang-tidy` may require header files to parse C++ files. We found that `clang-tidy` fails to parse C++ files in GCC-compiled projects (e.g., PHP) where the required header files contain GCC extensions like `asm goto`. `elm-lint` may also fail to parse files due to irregular comments. In contrast, our approach was robust to syntactic irregularities.

Clojure. To evaluate expressiveness more deeply, we implemented multiple patterns based on a dedicated syntax rewriter for Clojure. `Kibit` replaces Clojure code patterns with more idiomatic or terse versions (cf. `simpl-check` in Table 2.1). `Kibit` is an apt tool for comparison because it uses logic programming to specify rewrite patterns that are syntactically close to Clojure syntax. For this reason, we were able to easily translate `Kibit` rewrite patterns into our match and rewrite template format (some are identical, modulo

hole syntax). We implemented 81 templates for simplifying arithmetic, equality, collection, and control flow syntax for Clojure. `Comby`'s output is identical to `Kibit`'s on `Kibit`'s tests for all of these patterns. We were unable to implement one pattern that `Kibit` supports, which relies on detecting class names for static methods versus object methods. `Kibit`'s logic programming pattern representation is comparatively short and closely matches our declarative syntax (many specifications fit on one line, which accounts for a lower number of LOC in Table 2.3).

2.5.3 Discussion

Experience compared to existing tools. We noticed the following undesirable behavior in existing tools: (1) failure to handle files due to parsing errors or inadequate parsing support (2) false negatives where tools miss rewriting opportunities (3) hanging on certain files. From a usability perspective, we found `Comby` to be more robust: we never encountered runtime parsing errors, were able to transform code that other tools could not parse, and performed transformations that other tools missed.

Known limitations. Our approach enables lightweight, purely syntactic rewrite patterns. We emphasize simple and declarative specification, and have shown that patterns are particularly effective at, e.g., removing redundant code. However, we do not currently incorporate type information, which limits our ability to implement richer rewrite patterns found in tools like `clang-tidy`. This is problematic for languages with syntactic ambiguity with respect to program manipulation [156], as we noted for the Go `nil-check` pattern. However, this problem is not unique to our approach: linters can suffer false positives for the same reason. Further, redundant syntax, like redundantly parenthesized expressions, may require additional match templates (or rules) to account for syntactic variation.

For transformations that change nesting levels or whitespace, code may need reindentation. We treat whitespace liberally by default; enabling finer-grained control over whitespace

complicates matching on non-whitespace syntax. We found that patterns sometimes do preserve desired whitespace, but in general a post-process formatting tool is effective for (re)applying whitespace style.

In our experience, declarative templates are well suited to patterns that describe a syntactic change that does one thing well. More complex transformations may be difficult to express in a single rewrite pattern, and could require running transformations in succession (consistent with work using rewrite stages for transformation [174, 185, 194]). There may otherwise be syntactic transformations that are overly difficult or impossible to specify declaratively. Such patterns are hard for us to qualify generally; we mitigate by having shown real-world value using existing patterns.

We have not yet implemented support for layout-sensitive or context-sensitive language properties. Our approach therefore cannot handle indentation-sensitive properties of languages like Python. Specifically, it is not possible to match on nested code blocks that are delineated by whitespace. Parser combinators are sufficiently powerful to recognize these constructs, and we thus believe our approach can extend to handle this complexity. For example, we could emulate support by transforming Python code to a grammar with explicit block delimiters as in the `indent.py` script. Declarative templates are correspondingly converted to a representation that matches a correct level of *relative* indentation.

On the other hand, it is not always possible to recognize certain constructs in statically typed languages (like OCaml) based on syntax alone. Thus, our approach can fail to match constructs corresponding to, e.g., the complete syntactic extent of an expression following a `let` binding in OCaml. The possibility of incorporating type information may reveal ways to overcome these challenges.

2.6 Related Work

Parsing expression grammars (PEGs) [90] were developed to address shortcomings in regex matching, and can match non-regular constructs (e.g., balanced parentheses). PEGs describe a top-down parser for a language using the ordered choice operator. Existing tools implement PEGs to enable non-regular matching (`Rosie` [36], `instaparse` [32]) and modular syntax extension (`Rats!` [101]). Match specifications for these tools resemble grammar definitions and include, e.g., explicit metasyntax for parsing with ordered choice. Our match approach can be seen as automatically generating PEG-like parsers from declarative match templates, where templates lack metasyntax besides holes, and abstract from the underlying grammar interpretation.

`TXL` [73] is a transformation tool applicable to multiple languages where users specify both a grammar specification and transformation rules in the `TXL` language. Learning the `TXL` language (where, e.g., the grammar nonterminals are typically referenced or defined in the rewrite specification) is expensive compared to our approach. Similarly, language workbench tools [88] like `Rascal` [125] expose metaprogramming languages with grammar metasyntax to enable transformation. The `srcML` [71, 72, 150] platform features transformation for multiple languages, but uses an intermediate XML representation requiring programming to manipulate syntax, rather than declarative specification. `Cobra` [110, 111] is a lightweight approach for implementing syntactic checks (and optional transformation) primarily through programming and a query language.

Compared generally to these approaches, our design disentangles match and rewrite specification from grammar definition through a predefined (but configurable) DEL grammar operationalized as a parser generator. Template specification is declarative as opposed to programmatic. Matching is a function of parser generation that acts directly on syntax, without intermediate tree representations. This notably removes the complexity of translating a declarative concrete syntax to an underlying abstract syntax.

`Cubix` is a multi-language transformation tool using compositional data types and is a heavyweight solution intended for complex rewrites (like lifting variable declarations) with strong, type-based guarantees [127]. Our approach differs in two principle ways. First, we prioritize lightweight transformation (templates take seconds or minutes to write), whereas `Cubix` is not intended for lay programmers. Second, Cubix intends to amortize the cost of defining transformations across multiple languages. In principle, a developer writes a transformation pattern just once (e.g., to perform variable hoisting) which can then be performed on programs for many language at once. In contrast, our approach intends to be generically declarative, but sensitive to individual language syntax.

`Coccinelle` [132] is notable for using a declarative approach with similar specifications to ours to transform syntax, although it introduces additional metasyntax and uses a fixed grammar that only parses C code. `Refaster` [203] introduces a template-based approach restricted to Java, but can additionally draw on type information. D-expressions derive from Lisp macros [86] and use a skeleton syntax tree to support a simple core grammar definition for the Dylan language [45], later adapted for Java [46]. In our work, an extensible skeleton parser encodes handling syntactic variations to enable macro-like functionality for many languages.

Parser combinators [112] are central for enabling our modular transformation approach for flexibly handling multiple language syntax. In this sense, we relate to multi-language static checking using parser combinators [57]. Our focus, however, is on transformation. Partial or selective parsing as in semi-parsing [214], island-grammars [159], and micro-grammars [57] relate to our idea of using a coarse grammar, which we parameterize with respect to language-specific syntax. The relation between structural tree matching and parsing is observed by [109]; in this sense, our approach operationalizes configurable tree-like matching as a function of modular, configurable parser combinator generation via declarative specification.

Okasaki observes the phenomenon of parsers-producing-parsers [171] and notes how a

"prelude" parser can define another parser for parsing the remainder of a program. Templates in our approach correspond to a kind of extensible "prelude" parser; the resulting parser runs on the complete program and records only syntactic fragments of interest for rewriting.

Our approach extends Dyck languages (being simple yet fundamental to CFL properties [89, 94]) as a foundation for representing contemporary language syntax and structure. Visibly pushdown languages [41] also build on the essential CFL properties of Dyck languages toward term rewriting [62] and program analysis [64].

Refal [35, 194] is a pattern matching language and tool for supercompilation. Refal's operation and application emphasize our own: matching and rewriting patterns is valuable for removing redundancy and improving performance. Our tools presents further potential for large-scale cleanup and code changes, such as those performed by Google's `Rosie` [138].

2.7 Summary

This chapter presented a lightweight approach to syntactic transformation for multiple languages. In general, defining a universal AST for transformation is problematic: concrete definitions must be referred to, modified, or extended to support multiple languages. Our approach skirts this issue by focusing on universal CFL properties of languages (i.e., nested constructs) rather than defining a concrete representation. We lift the matching problem to declaratively generate parsers that detect syntax of interest directly. We use PPCs to modularize the behavior of parser generation to (a) preserve essential CFL properties while (b) allowing hooks to customize the parsing of language-specific syntax. Matched syntax is recorded during parsing and transformed contextually and hierarchically based on rewrite templates.

We implemented these ideas in a new tool called `Comby`. We evaluated our approach by applying 30 rewrite patterns to the 100 most popular GitHub projects for 12 languages totaling 185 million lines of code. Over 50 changes have been merged into upstream

repositories. We also showed that our declarative approach requires significantly less effort to specify rewrite patterns compared to language-specific tools used in practice. Ultimately, having a flexible way to manipulate syntax selectively across languages is valuable for techniques in automated program transformation. In subsequent chapters, we show how this ability allows new ways to tailor programs to achieve better analysis results (Chapter 3, Section 3.3.1 and Chapter 4, Section 4.4.2) and overcoming syntactic barriers to performing fixes in automated program repair (Chapter 5, Section 5.3.3).

Chapter 3

Tailoring Programs for Static Analysis

This chapter presents a new technique for tailoring programs to improve static analysis. The key idea is to apply program transformations (using templates as in Chapter 2) that target problematic code patterns that are known to induce false positive bug reports. Our technique promotes program transformation as a general primitive for improving the fidelity of analysis reports (we treat any given analyzer as a black box). Our evaluation is the first systematic study to broadly demonstrate the applicability and benefits of this technique and perspective: we evaluate false positive reduction for five different static analyzers supporting three different languages (C, Java, and PHP) on large, real-world programs (>800KLOC). We show that our approach is effective in sidestepping long-standing and complex issues in analysis implementations.

3.1 Introduction

Static analysis has proven indispensable in software quality assurance for automatically catching bugs early in the development process [83, 186]. A number of open challenges underlie successful adoption and integration of static analyses in practice. Analyses must approximate, both theoretically [129, 142] and in the interest of practicality [61, 68]. Devel-

opers are sensitive to whether tools integrate seamlessly into their existing workflows [116]; at minimum, analyses must be fast, surface few false positives, and provide mechanisms to suppress warnings [67]. The problem is that the choices that tools make toward these ends are broadly generic, and the divergence between tool assumptions and program reality (i.e., language features or quality concerns) can lead to unhelpful, overwhelming, or incorrect output [114].

Tool configuration and customization is crucial for usability and directing analysis behavior. The inability to easily and selectively disable analysis checks and suppress warnings present a significant pain point for analysis users [67]. Common existing mechanisms include analysis options for turning off entire bug classes (generally too coarse [116]) or adding comment-like annotations for suppressing spurious warnings at particular lines (a predominantly manual exercise that leads to code smells and is insufficiently granular [67, 116]). It is notably the *analysis author*, not the user, who has agency over the shape of these analysis knobs: which configuration options are available and how to suppress errors. It follows that it is infeasible for analysis writers to accommodate individual user preferences or analysis corner cases through a myriad of configuration options or suppression mechanisms.

An analysis *user* always has one notable ability to influence analysis behavior and output: modifying their program. For example, developers may slightly modify existing code in a way that suppresses false positives or undesired warnings. A concrete example is shown in Figure 3.1 where a change was made in rsyslog[1] to suppress a Clang Static Analyzer warning. The analyzer warns that a potential out-of-bounds access occurs when comparing two strings using `strcmp`. However, the warning only surfaces when complex macro expansions take place (in this case, macro expansion activates for `strcmp` because it is passed a string literal). One contributor notes that under normal circumstances these warnings are suppressed for macros, but can surface if macro preprocessing is done

[1] https://github.com/rsyslog/rsyslog/commit/ea7497

```
+ // clang static analyzer work-around
+ const char *const const_END = "END";
+ if(strcmp(rectype, const_END))
- if(strcmp(rectype, "END"))
```

Figure 3.1: A program change that works around a static analysis issue in `rsyslog`.

manually.[2] The workaround in Figure 3.1 extracts the string literal out of the `strcmp` call so that no macro expansion takes place. After the change, the analyzer sees `strcmp` as a C library function and no longer emits a warning. In practice, modifying programs to avoid analyzer issues is exercised as manual, one-off changes. Changes like the one above can have the undesirable side effect of persisting in the code solely to suppress unwanted analysis warnings. Despite these shortcomings, modification of the analysis target (the program) does, however, allow a primitive for *software authors* to draw on their particular domain knowledge of their code to positively influence analysis behavior. For example, a developer may recognize that a particular API call is the cause of a false positive warning, and modify or model the call differently to suppress a false positive.[3]

Our technique in particular affords *analysis users* the ability to change and suppress analysis behavior when no recourse is available via analysis support (e.g., when analysis maintainers delay fixing analysis issues, or limit tool configuration options). Our insight is that the same flavor of one-off, manual code changes like the one above can apply generally and automatically to remedy analysis shortcomings. Additionally, workaround changes need not have the undesirable trait of persisting in production code, and are applied only *temporarily* while performing analysis. We propose a rule-based approach where simple, declarative templates describe general syntactic changes for known problematic code patterns. Undesired warnings and false positives are thus removed during analysis by rewriting code fragments. Our approach can be seen as a preprocessing step that tailors programs for analysis using lightweight syntactic changes. It operates on the

[2] https://bugs.llvm.org/show_bug.cgi?id=20144
[3] https://github.com/facebook/infer/issues/781.

basis of *temporary* suppression (a desirable trait in configuring analyses [116]) and also enables catching false positives *before* they happen by rewriting problematic patterns. Since patterns can occur across projects, codifying transformations as reusable templates amortizes developer effort for suppressing false positives.

The notion that syntactic transformations abstract semantic transformations [75] underpins our intuition that manipulating syntax can achieve desirable changes in the analysis domain and implementation. Work on semantic properties of transformations emphasize the potential for improving analysis precision [144, 162]. Despite theoretical appeal for tailoring analyses via transformation [162], there is currently little demonstrated applicability or benefit in practice.

A key objective of our work is to demonstrate the broad feasibility, applicability, and effectiveness of these ideas for the first time in practice. To this end we evaluate on large, real-world programs written in a variety of popular languages. A significant challenge lies in recognizing and transforming syntactically-diverse languages for such an approach to work. Our technique for declarative multi-language transformation in Chapter 2 addresses this challenge and forms the basis for operationalizing our approach. We address analysis issues broadly by considering (a) user-reported false positives across multiple active analyzers and (b) historic user commits for suppressing analysis warnings. We develop useful transformations that best tailor to real analyzer issues at hand. Transformations can thus be semantics-preserving or semantics-altering, address shortcomings or oversights in analysis implementation, or improve analysis precision by altering how functions are modeled. We demonstrate the effectiveness of our approach by contributing the following:

- We operationalize the process of tailoring programs for static analysis using declarative syntax rewriting.
- We develop 16 transformation templates for improving the analysis output of five modern and active analyzers (`Clang`, `Infer`, `PHPStan`, `SpotBugs`, and `Codesonar`) We

show how our approach resolves real (including yet-unresolved) false positive issues affecting users.

- We demonstrate our approach on 10 real-world projects (including large ones, >100KLOC) across three languages: PHP, Java, and C.
- We show that our approach is efficient. Transformation typically takes one to three seconds (compared to analyses that typically take in the order of minutes); our approach is sufficiently performant to integrate into existing continuous integration and analysis pipelines.
- We show that our approach improves static analyses by reducing the number of false positives without adversely affecting other results. In aggregate we remove 107 false positive reports with a median of 2 false positives is removed per project.

3.2 Motivation

Figure 3.2a illustrates a past issue in `PHPStan`, a popular PHP analyzer. A file is opened in line 2, and assigned to a handle `$file` *inside* the `if`-condition on line 2. If opening the file fails, `$file` is assigned the value `false`, the condition evaluates to true, and the function immediately returns (line 3). On the other hand, if execution passes the check then `$file` is guaranteed to be valid (i.e., not `false`). The problem is that `PHPStan` does not track the effect of assignments in `if`-conditions, and reports an error saying that `$file` may be `false` when passed as an argument to `fputcsv` on line 7.

The issue for this false positive stayed open and unresolved for *over a year* on GitHub, and is cross-referenced in 13 related user-reported issues.[4] One project member responded that the issue is important and will be fixed in the future, but that “no one has yet figured out how to implement it without rewriting major part of the (analysis) core.” The fix imposed

[4]https://github.com/phpstan/phpstan/issues/647.

```
1  function test(): void {
2    if (($file = @fopen('file', 'wb+')) === false) {
3      return;
4    }
5
6    // analyzer complains that $file may be false
7    if (\fputcsv($file, [1,2,3]) === false) {
8      \fclose($file);
9      return;
10   }
11   ...
12   \fclose($file);
13 }
```

(a) Assigning the result of `fopen` to `$file` in Line 2 confuses the analyzer. It misses the effect that `$file` is not false on Line 7, and emits a false positive warning that `$file` may be false when passed to `$putcsv`.

```
1  function test(): void {
2    $file = @fopen('file', 'wb+');
3    if ($file === false) {
4      return;
5    }
6    ...
7  }
```

(b) An analysis user proposed this workaround: extract the assignment out of the conditional. This allows the analyzer to correctly track the assignment effect and does not emit a false positive. Unfortunately this approach is hard to blanket apply automatically and diverges from developer preferences who prefer this style for readability.

Figure 3.2: Variable assignment inside if-statements can cause a false positive report in `PHPStan`. A workaround is to put the assignment outside of the if conditional.

significant effort on analysis authors which delayed a resolution for months. Although some analysis authors were in favor of code that avoided assignments in conditionals, others found the style improves readability. Multiple users noted that the false positive can be avoided by a code change that extracts the assignment out of the conditional (Figure 3.2b).[5] However, one user also noted that this workaround was "a bit annoying" because it introduced redundancy. The proposed workaround also imposes a burden on the user to identify and refactor all affected instances.

Our approach introduces a new way to address these tensions. The intuition is that workarounds via code changes, as in Figure 3.2b, *can* generalize to cater for individual user preferences and overcome long-standing analysis issues. The high level idea is to write simple, declarative syntactic templates that can blanket apply automatically over an entire code base. Although the code changes *could* be persisted in the code, they need not be: our approach foremost promotes code changes as a temporary suppression mechanism with respect to a particular analysis. In our approach, a match template (as in Section 2.2) specifies a pattern that is syntactically close to the problematic pattern in Figure 3.2a:

```
if ((:[v] = :[fn](:[args])) === false)
```

A rewrite template replaces all instances of the match template, extracting the assignment out of the `if`-condition. The rewrite template is syntactically close to the pattern suggested by the user in Figure 3.2b:

```
:[v] = :[fn](:[args]);
if (:[v] === false)
```

The match template matches on the syntactic pattern where variable `v` is assigned the return value of calling a function `fn` with arguments `args`. As before, all other syntax is matched *concretely*; whitespace in the template matches all contiguous whitespace in the source code. For illustration we use this template to match on function call syntax because the analysis particularly tracks values for modeled functions like `fopen`.

[5] https://github.com/phpstan/phpstan/issues/1739

Using the above patterns we identified and rewrote 27 instances of the `if`-assign pattern in the WordPress and PHPExcel projects and removed 82 false positives due to issue #647 in `PHPStan`. Our approach has the positive effect of removing more false positives than matches, because the issue has a cascading effect of inducing false positives along multiple execution paths (we elaborate in Section 3.4).

Writing these declarative patterns is easy compared to implementing additional analysis reasoning, and sufficiently general to overcome analysis shortcomings. The format of syntactic templates is accessible to developers; indeed, templates can be written in a format that is syntactically close to user-identified and user-implemented workaround transformations (as in Figure 3.2b). In our experience, complex changes to an analysis implementation can have a correspondingly easy resolution via code transformation patterns; templates can be developed in minutes for issues that take days to months to resolve in an analysis implementation (or even issues without any proposed solution whatsoever).[6] Notably, our declarative approach allows to write simple replacement patterns that express nontrivial syntactic properties that go beyond the capabilities of regular expression search-and-replace. This ability enables the program tailoring approach, because we can target rich syntactic patterns and transform these toward the objective of improving analysis fidelity.

3.3 Approach

This section explains the overall process of our program tailoring approach. Section 3.3.1 briefly reviews matching behavior as it pertains to templates in this chapter. Section 3.3.2 explains how we integrate program transformation for improving the quality of analyzer bug reports and the principles behind our approach.

[6] See, e.g., https://github.com/spotbugs/spotbugs/issues/493

3.3.1 Match Template Behavior

We uses `Comby` for declaratively rewriting syntax with templates (cf. Chapter 2). Matching behaves as follows:

- The `:[hole]` syntax binds matched source code to an identifier *hole*. Holes match all characters, including whitespace, lazily up to its suffix, but only *within* its level of balanced delimiters.
- `":[hole]"` (inside string quotes) matches strings within those quotes.
- `:[[hole]]` matches only alphanumeric characters in source code, similar to `\w` in regex.
- All non-whitespace characters are matched verbatim.
- Any contiguous whitespace (spaces, newlines, tabs) in a match template matches any contiguous whitespace in the source code. Match templates are thus sensitive to the presence of whitespace, but *not* the exact layout (the number of spaces do not need to correspond exactly between match template and source code).
- Matching is insensitive to comments in the source code. Comments are treated like whitespace when matching non-hole syntax in the match template.

In addition, we use rules to refine match and rewrite conditions. As before, rules specify optional predicates on matched syntax, such as equality of variables. We elaborate on rules as needed in the rest of the chapter. The match template, rewrite template, and rule comprise the full input for a single transformation. Each part is passed on the command-line.

3.3.2 Tailoring Programs for Analysis

This section explains our approach to tailoring programs for analysis. Figure 3.3 illustrates the main phases in our approach; we use it to characterize the process in detail.

Who writes templates? Our approach starts with a manual step where users ❶ write transformation templates. A key principle of our approach is to allow particularly *analysis*

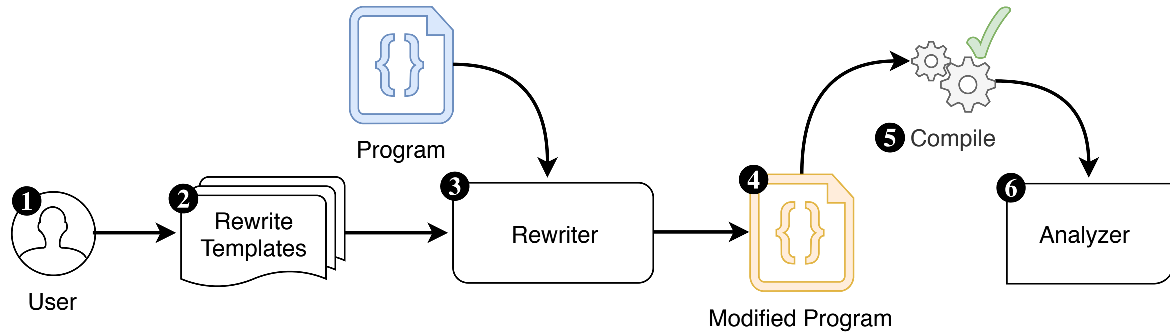

Figure 3.3: Overview of the program tailoring process.

users to influence static analyses via program transformation. The simplicity of declarative templates makes syntax transformation accessible, so that user observations and workarounds (as in Figure 3.2b) can be straightforwardly implemented. This primitive is not exclusive to analysis users, but also accessible to analysis writers and external contributors. For example, analysis writers can create and distribute transformation templates for issues on an interim basis (such as for Figure 3.2a). Users can then apply these templates as a temporary workaround before analyzing their projects. In this dissertation, we (the author) develop and evaluate transformations as an external contributor—we treat analyses as a black box (as a user would) and develop transformations for extant issues in popular analyzers (as an analysis writer might).

What properties do templates have? Writing rewrite templates ❷ is a one-off exercise per transformation. Templates are generally short (we develop templates that are between 3 and 15 lines long). Our results show that there is typically a one-to-one correspondence of rewrite template to analysis issue, where an analysis issue can entail a general problem in analysis implementation, code generation, or function modeling. Formulating rewrite templates requires competitively low effort compared to implementing complex changes analysis-side.

Templates are customizable. For example, we can refine the template `if ((:[v] =`

`:[fn](:[args])) === false)` to match on a particular concrete function, such as `fopen`, instead of `:[fn]`. Rudimentary mechanisms (e.g., C macros, comments, or assertions [87], and bug auto-patching [4]) have been used and recommended for suppressing false positives in existing analyzers [3, 6]. These mechanisms suffer from relying on language-specific features, are brittle and coarse, and built into the tool or hardcoded in the program. Our template-based mechanism is broadly accessible (it is easy to write templates), customizable, and generic across languages for manipulating syntax. It operates independent of language toolchains, does not presume the availability of language-specific features (e.g., macros), and does not prescribe any configuration for external analysis tools or compilers. Templates currently express purely syntactic properties (we cannot, e.g., draw on type information to inform manipulation). However, future work can incorporate richer semantic information (Section 6.2.5).

A set of templates form a catalog of transformations that can be reused across projects that use analysis. A catalog is the starting point for an automated pipeline, driving the remaining phases in Figure 3.3. In this chapter, we present a catalog of human-written templates that directly targets known analysis issues. We note the especially interesting potential for automating template generation in ❷ (e.g., by mining code patterns for inferring templates), which we discuss in future work (Section 6.2.3).

What are the principles behind automated code changes tailored to analysis? The rewriter ❸ takes as input the set of rewrite templates and a program. It rewrites matched syntax in the original program to produce a modified program that will be analyzed. The modified program is intended to be a *temporary* representation of the original program that is discarded after analysis. In practice, users can identify transformational workarounds to issues (as in Section 3.2), but may not want to persists those changes in their code.[7] We implement temporary program modification by running the rewriter on a version controlled

[7] A different user experience states: "Changing the property gets rid of the false positive, but that's not what I want" – https://github.com/Microsoft/CodeContracts/issues/255.

project (we use `git`). After running the analysis and capturing the bug reports, the project is reverted to its original state. Syntactic rewriting is fast; less than 4 seconds to process an 800KLOC project, and introduces negligible overhead compared to analysis runtime.

This chapter considers transformations primarily as a targeted suppression mechanism for analysis false positives. However, we observe that transformations can also be tailored to surface additional *true positive* bugs (e.g., as an effect of performing automated program repair, which we consider in Section 3.4).

The rewriter can be seen as a preprocessor for analyses. One principle of running our approach *prelude* to analysis is that we can dually use templates to search for and detect (but not necessarily rewrite) problematic syntactic patterns. This mechanism provides an early smoke signal for patterns that may trip analyzers *even before the analysis runs*. We notably reuse our templates to detect false positive-inducing patterns across projects in our evaluation. Large scale efforts can similarly detect the extent of possibly affected projects when adopting a new analyzer; analysis writers may use it to prioritize analysis fixes. These capabilities are notable in contrasting our approach to existing mechanisms. Syntactic templates are valuable in explicitly codifying sensitivities of analysis behavior as it relates to program structure, whereas suppression-by-comments and configuration options provide an escape hatch in anticipation of problematic analysis interactions where program structure is readily ignored.

Automated program transformation is a powerful primitive, and applying incorrect templates can produce malformed programs. When developing templates, it is helpful to impose validation criteria for running the analysis on the modified program.

What are the conditions for analyzing a modified program? Analyses generally accept only well-formed programs (in the respective language), and typically rely on building artifacts or instrumenting compiler output to perform analysis. Language allowing, we impose a validation step that all modified programs must compile ❺ when performing

analysis, which provides additional assurance that the analysis will not terminate early due to malformed programs. For PHP, we rely on a valid parse of target files as the validation step. Applying transformations may violate style checkers (e.g., linters) integrated into a build manifest, or cause spurious compiler warnings (e.g., unused variables), but still allow the program to compile successfully. We allow such violations or warnings; in our results, these do not directly affect the fidelity of the analysis with respect to the issue targeted by transformation. Another possibility of interference is that transformations addressing one bug class may report bugs in a different bug class (e.g., a transformation may introduce a dead store). This undesirable behavior depends on the analysis checks, its configuration, and template formulation. We qualify such cases during our experiments concretely in Section 3.4.

In the final step we capture analyzer output for the modified program ❻. The goal of our approach is that this output provides a higher quality bug report than that obtained from running the analyzer on the original program.

3.4 Evaluation

This section describes our results tailoring programs for analysis. Our evaluation emphasizes real-world applicability on large and popular programs for modern analyzers. Section 3.4.1 describes our experimental setup. Section 3.4.2 describes our results applying 16 rewrite templates to 10 projects across 5 analyzers. We discuss further considerations and limitations in Section 3.4.3.

The goal of our evaluation is to show that programs *can* be tailored generically and declaratively to improve analysis output. We focus on breadth of applicability across languages and analyzers for real-world issues. Thus, our research questions are:

RQ. 1 **Can tailoring programs improve the fidelity of static analysis reports?**
Specifically: Does the approach remove false positive reports without otherwise

Analyzer	Lang	Pattern	Project	KLOC	Time		Issues			#R	Ref
					Anlyz	Rewr	Bef	Cls	Δ FP		
PHPStan	PHP	if-assign-resources	WordPress	5.7	2.5s	0.3s	112	44	-44	16	647
			PHPExcel	5.1	0.7s	0.1s	113	38	-38	11	
		substr-model	dompdf	3.9	0.5s	0.2s	34	1	-1	1	1215
			matomo	0.3	0.2s	0.1s	21	1	-1	1	
			PHP_CodeSniffer	1.6	1.1s	0.1s	123	3	-3	3	
			neonizer	0.1	0.1s	0.1s	1	1	-1	1	
Infer	C	free-model	OpenSSL	402.9	17m29s	1.6s	462	22	-6	3,735	120
	Java	create-socket	Drift	50.7	2m47s	1.0s	62	1	-1	1	781
			Presto	813.0	39m20s	3.3s	822	106	-1	1	
		wrapped-resources	hazelcast-jet	103.8	5m24s	1.3s	29	7	-2	16	999
SpotBugs	Java	null-on-resources	hazelcast-jet	103.8	1m48s	2.0s	1	1	-1	45	756
			Santulator	11.1	44s	0.5s	2	2	-2	8	
Clang SA	C	const-strcmp	rsyslog	145.0	16m20s	2.9s	3	3	-3	10	ea74
CodeSonar	C	strncpy-null	swoole-src	96.8	6m40s	0.6s	117	1	-1	1	—
		snprintf-null	ioping	1.2	0m14s	0.2s	12	2	-2	2	

Table 3.1: Main results of our approach. Each row represents an application of a transformation **Pattern** on a project. **Issues, Bef** is the total analyzer warnings before transformation. **Cls** is subset of total analyzer warnings in the bug class that the transformation targets. (**Δ FP**) is the number of warnings removed by the transformation for the targeted bug class. **Ref** is the external GitHub issue or commit reference. **Time, Anlyz** is the time to run the analyzer on each project (given in thousands of lines of code, **KLOC**). **Rewr** is the time to process and rewrite each matching pattern across **KLOC**. We remove 107 false positives (**Δ FP**) in total, with a median of 2 per project.

adversely affecting the analysis results?

RQ. 2 **Does the program tailoring approach generalize?** Specifically, we evaluate generality with respect to multiple languages and analyzers, and pattern reuse across projects.

3.4.1 Experimental Setup

We consider five popular analyzers: `PHPStan` [7] for PHP, `SpotBugs` [8] for Java, `Clang Static Analyzer` [1] and `CodeSonar` [2] for C, and `Infer` [5] for both Java and C. All analyzers are mature, actively used, and incorporate state-of-the-art techniques. All analyzers are open source, except for `CodeSonar` which is a commercial analyzer.[8] For each analyzer we were interested in current or long-standing issues that cause spurious warnings or false positives, and particularly issues that could not be easily addressed by analysis configuration or suppression mechanisms:

1. We searched the `PHPStan`, `SpotBugs`, and `Infer` issue trackers on GitHub for reports or comments containing the words "false positive".
2. We found related issues for the `Clang Static Analyzer` (which does not have a GitHub issue tracker), by searching for GitHub commits containing the words "clang static analyzer false positive".
3. `CodeSonar` does not have a public issue tracker, nor did we find public commits referencing false positives. We manually inspected warnings for false positives.

The above methods influenced project selection. For (1) we identified problematic syntax patterns and user-affected projects from user reports. For (2) we identified committed code changes in an existing project and used this to develop a generic template. For (3) we selected popular C projects as representative real-world projects (since we did not have sources indicating false positives in `CodeSonar` a priori).

[8]We used `CodeSonar` under a free academic license.

The templates we developed from these sources can apply generally across projects. We thus reused our templates to search for additional projects containing potential false positive-inducing patterns in a large-scale search. We searched over the top 100 most popular[9] projects for each language. However, to apply our approach generally, we require that a project (a) compiles (or parses) successfully and (b) is configured for analysis. Many of the projects identified by large-scale search presented significant manual burden to compile (e.g., due to various dependencies and platform-specific requirements like Android, iOS, or Visual Studio) and consequently configure for analysis. We note that this burden is amplified for external users (such as ourselves) who have limited access to various platforms and who are unfamiliar with specific build configurations. We generally expect the burden to be less of an impediment when our approach is used by project maintainers, who are familiar with the complexities of their own projects.

Our selection ultimately represents a convenience sample of real-world issues to substantiate our claims about tailoring programs to overcome analyzer limitations. The selection includes issues that affect real developers of large, popular codebases. We show that our approach is efficient and scales to these concerns, and that it presents potential for reuse across projects through large-scale search.

We ran our large-scale search experiments on an Ubuntu 16.04 LTS server, with 20 Xeon E5-2699 CPU cores and 20GB of RAM. We evaluated analysis improvement on the same for large projects (>100KLOC) and the `CodeSonar` analyzer. All other analysis improvement experiments were run on an Ubuntu 16.04 VM with two 2.2 GHz i7 CPU cores and 4GB RAM.

[9]Ranked by the number of user favorites (GitHub stars) in July, 2019.

3.4.2 Experimental Results

Table 3.1 shows our results for each analyzer. We develop transformations for 16 syntactic **Patterns** across PHP, Java, and C projects. The **Issues** column shows the total number warnings across all bug classes in the **Bef**ore column (each analyzer is run with its default checks). We ran each analyzer on the entire project (warnings are thus for the entire project) except for `PHPStan` where the number of warnings is reported for a single file. `PHPStan` can operate at the file level, and using targeted transformation we demonstrate that our approach can isolate issues in individual files without incurring a project-wide analysis. The **Cls** column is the subset of all bugs that fall into the bug **Class** that the pattern targets. The Δ **FP** column is the number of false positives removed for that bug class by transformation. The #**R** column is the number of rewritten instances in the source code for each pattern. There may be more or fewer rewritten instances compared to removed false positives due to the effect of transformations on analyses (cf. pattern **free-model** and **if-assign-resources**); we elaborate below. The time to transform the program (**Time Rewr**) is negligible compared to analysis runtime (**Time Anlyz**). In aggregate these transformations remove 107 false positives (Δ **FP**) out of a total of 253 reports within the targeted bug class (not all of which are false positives). A median of two false positives are removed per project.

Effects of transformations on analysis behavior.

Providing a primitive for arbitrarily modifying programs means that our approach can hypothetically introduce adverse effects which do not exist in the original program. The negative possibilities are that a change either removes true positives in the original program, introduces more false positives in the modified program, or both. We evaluated whether our patterns cause such adverse effects by manually inspecting bug reports before and after applying a transformation for every analysis run. From our inspection, no true

positives are removed for any project. No additional bug reports are introduced for any project, except OpenSSL and Presto. Infer nondeterministically reports different numbers of bugs for large Java projects in the case of Presto, irrespective of whether we perform a change. We confirmed that our change removes the false positive on five independent runs; nondeterminism for large numbers of bug reports (>800) make it difficult to conclude whether our change has any meaningful effect on other reports.

We observed that Infer reports an additional 6 potential null dereferences after applying the **free-model** pattern. This is interesting because the **free-model** pattern targets memory leak false positives, not null dereference reports. Interestingly, Infer analyzes and reports bugs in functions after transforming the program, whereas it previously short-circuits analysis and reporting for functions containing `free`-wrapper functions. We inspected these reports and could not obviously discern whether they were false positive or true positive reports.

In summary: Our approach removes false positives without adversely effecting the analysis in the majority of cases (13 out of 15 projects), while two projects are inconclusive. In general, program tailoring is effective because transformations perform small, local changes that affect only the reasoning of the analysis for that instance or bug class. It follows that changes which are closely semantics-preserving of the original program ought not make an analysis less precise, and our results affirm this intuition.

Amortizing human effort for codifying and detecting patterns.

Pattern reuse is an especially appealing property of tailoring programs. We developed four patterns from user-reported issues in a single project[10] which we then used to detect and fix multiple false positive issues in six additional independent projects. These results show that patterns may generalize to benefit multiple projects, and imply that the cost and human effort of writing broadly applicable templates can be amortized across software

[10]**if-assign-resources**, **substr-model**, **create-socket**, and **null-on-resources**

Language	MLOC	Pattern	# Matches	Time
PHP	8.9	if-assign-resources	42	18s
		substr-model	6	20s
Java	16.2	create-socket	1	56s
		wrapped-resources	254	30s
		null-on-resources	4	35s
C	29.4	free-model	52,461	7m
		const-strcmp	91	48s
		strncpy-null	1	48s
		snprintf-null	13,763	50s

Table 3.2: Large scale search results over the top 100 most popular PHP, Java, and C repositories on GitHub. **MLOC** is the millions of lines of code searched; **# Matches** is the number of matching instances across all projects. Search is fast and scales to millions of lines of code for rich patterns.

stakeholders (i.e., both analysis users and analysis writers develop, distribute, and benefit from patterns). In contrast, existing mechanisms using comment suppression or command line options cannot likewise generalize, and consequently induce recurring developer effort.

We performed a large-scale search using each pattern to identify the additional projects above, and to quantify overall efficiency.[11] Table 3.2 shows our results running each pattern on the top 100 most popular GitHub repositories for PHP, Java, and C. In general search is fast and can identify potential false-positive inducing patterns before an analysis even runs.

The accuracy of this approach depends on the template specificity, analysis behavior, and project-specific conventions. For example, we saw that the **free-model** pattern works well for the OpenSSL project, which uses a convention of custom `free`-wrapping functions. Naturally, this convention may not hold in other projects among the 52 thousand matches found by our search. On the other hand, we developed the **substr-model** using a user-reported issue for a single project (`neonizer`) and discovered five other matches in three additional projects, all of which cause real false positives in `PHPStan`, and which the transformation suppresses.

[11] Note that the patterns identified projects which failed to run under the analysis or compile, and hence not included in Table 3.1.

Fixed?	Issue	Duration	Reported	# Refs
✓	phpstan/647	1 yr. 3 mos.	11-2017	16
✗	phpstan/1215	1 yr. 1 mo.	07-2018	1
✗	infer/120	4 yrs. 1 mo.	06-2018	4
✗	infer/781	1 yr. 9 mos.	10-2017	0
✗	infer/999	10 mos.	09-2018	0
✗	spotbugs/756	11 mos.	09-2018	41

Table 3.3: Summary of false positive issues in active analyzers. A ✗ in the **Fixed?** colum indicates that the issue is still unresolved at the time of writing. Only one of the issues is currently resolved (✓). On average, the issues in our experiments have stayed unresolved for 1 year and 7 months (aggregated over **Duration** as of the date the issue was **Reported**). **# Refs** indicates the number of additionally cross-referenced issues for a report (including, e.g., duplicate user reports or external tools affected by this issue).

Issue duration and resolution for analysis end users

Table 3.3 characterizes six open source issues that our patterns address.[12] Interestingly, issues are long-standing (unresolved for over a year, on average), and all but one remain unresolved.[13] These issues generally reveal that analysis end users are subject to long delays and lack of support, having little recourse for resolving false positives using existing mechanisms. On the one hand, analysis writers may not have the time to support user- or project-specific needs, and may deprioritize less pressing requests (e.g., `infer/781` is an individual user request with no cross-references to other issues). On the other hand, an issue may affect many users (as shown by multiple cross-referenced issues for `phpstan/647`, `spotbugs/756`), but be very complex to solve for analysis maintainers.

Our approach gives agency to end users for implementing workarounds. Furthermore, user-reported errors can be significantly more severe than other analysis reports as they may break existing software workflows. For example, even a single false positive report in `SpotBugs` due to **null-on-resources** breaks the continuous integration (CI) build of cross-referenced projects, and caused users to disable the check wholesale for their projects across

[12]The three remaining patterns, **rsyslog**, and those for CodeSonar, do not have open source issues.
[13]In October, 2019.

Match template

```
if ((:[v] = @fopen(:[a])) == 0)
```

Rewrite template

```
:[v] = @fopen(:[a]);
if (:[v] === false)
```

Figure 3.4: The **if-assign-resources** pattern extracts a variable assignment out of an if-conditional (in this case for `fopen` calls).

all versions of Java.[14] Because of this profound effect, bug severity and build integration must be weighed into analysis configuration mechanisms. Moreover, although the relative size of false positive reduction is small for some reports in Table 3.1, the correspondence to real-world issues and extended impact on end users and software workflows make the false positives we handle more significant than other false positives. Our results show that our approach can uniquely address such complexities.

Rewrite patterns

We now discuss analyzer issues in greater detail, explain what our transformation does to resolve the issue, and why this transformation induces a positive change in analysis behavior.

Pattern: if-assign-resources. This pattern addresses the issue of variable assignment in `if`-conditionals (as introduced in Section 3.2). At the original time of writing `PHPStan` did not accurately track the effects of such assignments, and it took over a year to fix in the analysis implementation. False positives particularly manifest for cases where states of resources (like files) are opened. Because `PHPStan` does not track that the variable assigned cannot be false, it reports an error when the variable is passed to a function such as `fwrite` or `fclose`, saying "Parameter #1 of function `fclose` expects `resource`, `resource|false` given". The transformation pulls the assignment out of the if-conditional, which allows `PHPStan` to accurately track the effect (Figure 3.4). An interesting result is that rewriting this pattern can remove many false positive reports because multiple functions may use a

[14] See, e.g., https://github.com/confluentinc/ksql/pull/2144.

Match template

```
$:[[v]] = substr(:[a]);
if ($:[[v]] === false)
```

Rewrite template

```
$:[[v]] = substr(:[a]) || isset($maybe_false);
if ($:[[v]] === false)
```

Figure 3.5: The **substr-model** pattern informs `PHPStan` that a call to `substr` may return false.

file resource along multiple paths, and each use raises an error. For example, rewriting 16 cases in WordPress removes 44 false positives.

Pattern: substr-model. The `substr` function in PHP performs a substring operation. `PHPStan` models return value types of functions like `substr`. However, `PHPStan` did not track the fact that `substr` may return false if the length of the string is shorter than the requested substring range. `PHPStan`, thinking the value can only ever be a string, thus emits a false positive when a user's code checks whether the return value of `substr` is false, saying "comparison using `===` between string and false will always evaluate to false".

It took eight months as of the first user report for maintainers to implement the solution properly. It is particularly interesting that the maintainers were unwilling to make a simple change to the function model to reflect that `substr` can return false, because it would propagate spurious warnings when used as an argument in other contexts. However, users primarily had issues with the model when they *checked* return values, not when the value was used as an argument. Our transformation in Figure 3.5 matches problematic cases that users experienced while avoiding the difficulty of changing the `substr` model wholesale. We introduce the expression `isset($maybe_false)` which lets `PHPStan` reason that the return value of an unset variable may be false. Statements that assign the result of `substr` before an `if` check are changed with an or-clause, which effectively remodels the `substr` call to possibly return a `false` value. Note that the use of `:[[v]]` in the assignment statement (line 1) and conditional expression (line 2) of the match template introduce a constraint that both matching instances must be syntactically *equal* for the rewrite rule to fire. This pattern removes six false positive reports in four PHP projects.

Match template and rule

```
:[[f]](:[args])
```

```
where match :[args] with
| ":[_],:[_]" -> false
| ":[_]" -> true
```

Rewrite template and rule

```
:[[f]](:[args])
```

```
where rewrite :[f] with
| ":[_]free" -> "free"
```

Figure 3.6: The **free-model** pattern renames custom `XXX_free` functions to just `free`. It first matches all function calls, then filters calls that have only one argument using the `where match` rule. It then rewrites satisfying calls that contain the letters `free` using the `where rewrite` rule.

Match template

```
:[[sock]] = :[_].createSocket(:[[sock]], :[_], :[_], true);
```

Rewrite template

```
:[[sock]].close();
:[[sock]] = new Socket();
```

Figure 3.7: The **create-socket** pattern removes an unmodeled `createSocket` constructor that wraps a socket, and replaces it with an explicit, modeled sequence where the socket is closed and created again. The last argument in the match template (which implies the socket will be closed automatically) must be true in the original source for this rule to fire.

Pattern: free-model. `Infer` may timeout when analyzing functions and fail to summarize their effect. `Infer` reports false positive memory leaks when it fails to realize that a function frees memory—this happens particularly when the C library `free` call is wrapped inside custom free functions. The `OpenSSL` project follows the convention of wrapping free calls in functions like `OPENSSL_free`, which Infer fails to analyze. As an approximation of these custom free functions, our transformation (Figure 3.6) rewrites the wrapping functions to call the plain C library `free` version. OpenSSL compiles successfully despite transforming 3,583 calls, and removes 25 false positives because Infer can then track the effect of `free` on memory. The remaining memory leaks are unrelated to `free` calls.[15]

Pattern: create-socket. The `createSocket(socket, ...)` call is typically used in conjunction with the Java SSL library. It returns a server socket that wraps an existing `socket`

[15]These are also likely false positives, though we did not investigate fully.

Match template

```
  try (:[[o]] :[[v]] = new :[[c1]](new :[[c2]](:[args])))
```

Rewrite template

```
:[c2] wrapped = new :[c2](:[args]);
try {
  wrapped.close();
} catch (IOException e) {}
try (:[o] :[v] = new :[c1](wrapped))
```

Figure 3.8: The **wrapped-resources** rewrites the Java try-with-resources pattern where constructors wrap a file descriptor. The pattern closes the inner resource so that Infer stops tracking it (and stops it from reporting a leak.

in the first argument. The last argument, when `true`, tells the call that the underlying socket should be closed when the returned socket is closed. Infer fails to model the `createSocket` call, and a false positive report states that the underlying socket is leaked. The boolean toggle in the last argument makes this function difficult to model generically.

Interestingly, a user reports a false positive for a particular case where the last argument is always `true`. Our transformation (Figure 3.7) addresses the issue by simulating the close operation early, and simply returning a fresh socket for `Infer` to track. Note that we would never persist this change in practice as it loses the implementation details of `createSocket`; however, it is sufficient for making the analysis more precise. This pattern removes two false positive reports in two Java projects; no resolution has yet been proposed in the Infer issue tracker.

Pattern: wrapped-resources. Java 7 introduced the try-with-resources statement which automatically closes a resource after the block executes, preventing a leak. Infer reports a false positive leak when a resource constructor is nested inside another resource constructor within a try-with-resources statement (this happens, e.g., when passing a `FileInputStream` to an `InputStreamReader`). The underlying problem is similar to pattern **create-socket**: `Infer` fails to track that the wrapped resource will be closed. We use a conceptually similar transformation as in **create-socket**, but account for the syntactic variation introduced by

Match template

```
try (:[x] :[[v]] = :[expr]) {
  :[body]
}:[rest]
```

```
where match :[rest] with
| " catch" -> false
| " finally" -> false
| ":[_]" -> true
```

Rewrite template

```
:[x] :[v] = :[expr];
try {
  :[body]
} finally {
  if (:[v] != null) {
    :[v].close();
  }
}
:[rest]
```

Figure 3.9: The **null-on-resources** rewrites a Java try-with-resource statement to a more traditional try-with statement. This avoids a redundant null-check injected by the compiler in Java versions 11 and 12, which leads to an analyzer warning. The match rule ensures that matches pass only if there does not already exist a `catch` or `finally` clause after a try-with-resources statement.

try-with blocks (Figure 3.8).

This transformation suppresses two false positive instances in one Java project. A solution in the `Infer` issue suggests modeling the constructor classes to automatically close the wrapped resource. Unfortunately, this may be insufficiently precise and the issue remains unresolved.

Pattern: null-on-resources. `SpotBugs` reports a redundant null check on resources inside try-with-resources statements (i.e., a resource is null checked after being previously dereferenced).[16] However, the error is only reported for code compiled with Java 11 and 12, and *not* Java 10. The reason is pernicious: the Java compiler in later versions *inserts a null check in the bytecode* which does appear to be indeed redundant. From the user's perspective, however, the report is a false positive—no null check is visible in the source code. The issue is cross referenced by over 10 projects, and remains unresolved for over a year. Various projects have added annotations or disabled the check completely. No official solution has been proposed.

Our approach presents a fresh alternative using the transformation in Figure 3.9, which converts a try-with-resources statement to a traditional try-catch-finally block. In effect,

[16]Note this bug is orthogonal to **wrapped-resources.**

Match template

```
if(strcmp(:[1], ":[2]"))
```

Rewrite template

```
const char *const t1 = ":[2]";
if(strcmp(:[1], t1))
```

Figure 3.10: The **cons-strcmp** pattern rewrites cases of `strcmp` so that constant strings in the second argument do not trigger macro expansion, thereby suppressing the false positive report.

Match template

```
strncpy(:[dst], :[src], :[len]);
if (:[len] :[rest]) { :[dst][:[idx]] = 0; }
```

Rewrite template

```
strncpy(:[dst], :[src], :[len] - 1);
:[dst][:[len] - 1] = '\0';
if (:[len] :[rest]) { :[dst][:[idx]] = 0; }
```

Figure 3.11: The pattern **strncpy-null** pattern. This transformation fires when the destination buffer `dst1` has the same name as the buffer `dst2` that is null-terminated subsequently, and the conditional check using `len2` is dependent on the length `len1` used in the call.

we normalize the try-with-resources syntax across Java versions to sidestep the null-check generation that only happens for Java versions 11 and 12. This suppresses three spurious bug reports in two of the projects we evaluate; it is interesting to note that only a fraction of the rewritten patterns (3 out of 53 cases) in fact trigger the `SpotBugs` warning.

Pattern: const-strcmp. The Clang Static Analyzer may report a potential out-of-bounds access when comparing strings with `strcmp`. This only happens when macro-expansion (defined in `glibc` headers) is triggered, in this case by the fact that a string literal is passed in the second argument to `strcmp`. Although analysis writers try to suppress warnings raised by macro expansion, manual build steps for projects may bypass the suppression mechanism. The transformation in Figure 3.10 extracts the string literal in the comparison to a string `const`, causing the analyzer to see a `strcmp` C library function rather than analyzing the macro expansion.

Pattern: strncpy-null The C `strncpy` function does not necessarily null-terminate its destination buffer, which can lead to memory corruption. `CodeSonar` warns about this

```
1      static char *php_ssl_cipher_get_version(const SSL_CIPHER *c, char
       *buffer, size_t max_len) {
2        const char *version = SSL_CIPHER_get_version(c);
3  -     strncpy(buffer, version, max_len);
4  +     strncpy(buffer, version, max_len - 1);
5  +     buffer[max_len - 1] = '\0';
6        if (max_len <= strlen(version)) {
7            buffer[max_len - 1] = 0;
8        }
```

Figure 3.12: A warning is emitted at line 4, where `strncpy` may not necessarily null-terminate `buffer`. It is a false positive: the buffer is always null-terminated in the case where `max_len` characters are copied. Our transformation makes the null-termination explicit to suppress the warning.

possibility, but also notes that *if* a subsequent statement definitely null-terminates the string, then the warning can be ignored. We found that the warning was indeed a false positive in the `swoole` project: a subsequent check always null-terminates the buffer safely. However, a possible reason why `CodeSonar` conservatively reported an error is that the subsequent statement is guarded and was not considered safe (see line 10, Figure 3.12). To avoid the false positive, our pattern checks whether a condition on the `strncpy` buffer length terminates that same buffer with a null character (Figure 3.11). If so, the rewrite unconditionally null terminates the buffer. Although this transformation is not generally strong enough to match syntax that guarantees a null-terminated buffer, it does provide flexibility for refining analysis warnings. For example, we found exactly the same pattern using our template in the `PHP` source code, where it appears the `php_ssl_cipher`... function was borrowed from.

Pattern: snprintf-null `CodeSonar` reports an "Unterminated C String" error when a possibly unterminated string is passed to a function such as `strcat`. The analysis believes a string may not be null-terminated when, for example, space is allocated in the heap but not subsequently null-terminated. We identified two false positives where heap-allocated memory for a string *is* null-terminated, but only because we know that the buffer starts out with positive length and passes through `snprintf` which always null-terminates. The

Match template

```
snprintf(:[dst], :[len], :[rest]);
```

Rewrite template

```
snprintf(:[dst], :[len], :[rest]);
if (:[len]) > 0) {
  :[dst][:[len]] = '\0';
} else {
  :[dst] = NULL;
}
```

Figure 3.13: The **snprintf-null** pattern models the possibility that the `snprintf` destination buffer may be null if the length is zero (i.e., not positive.)

analysis introduces imprecision where it believes the string can be of zero length, but does not, however, model the possibility that the buffer can subsequently be treated as a null pointer when passed to `snprintf`. Our transformation (Figure 3.13) makes this result explicit, which suppresses two unterminated string warnings.

3.4.3 Discussion

This section further characterizes considerations and limitations of our approach.

Do transformations adversely affect results? Providing a primitive for arbitrarily modifying programs means that our approach can hypothetically introduce adverse effects (e.g., other false positives) which do not exist in the original program. In our results we did not observe adverse effects of our patterns. Most perform small, local changes that affect only the reasoning of the analysis for that instance or bug class. It follows that changes which are semantics-preserving, or closely semantics-preserving of the original program (e.g., **if-assign-resources**, **const-strcmp**, and **snprintf-null**) ought not make an analysis less precise, and our results affirm this intuition.

We observed that Infer reports an additional 10 potential null dereferences after applying the **free-model** pattern. This is interesting because we do not expect that changing `free` functions should affect nullable properties of pointers. It turns out that Infer can continue

analyzing functions after we transform the program, whereas it previously short-circuits analysis for functions containing `free`-wrapper functions. We did not confirm the validity of the additional null dereference warnings (as these are orthogonal to memory leaks), but note that transformations can improve analysis coverage (though not necessarily precision) for other bug classes.

True positive warnings in the modified program may appear at different lines compared to the original program. As a usability concern, affected lines in the modified program should map to those in the original. This is primarily an engineering concern, as transformations keep track of precise changes in offsets so that no information is lost.

Pattern development. We found that developing patterns can take a few iterations to refine until they precisely match syntax of interest. For example, we iteratively added constraints in the **null-on-resources** pattern to filter out `try` statements that already contained `catch` and `finally` statements (without this constraint we would generate malformed programs). We expect users of our approach to similarly develop patterns incrementally. This process is analogous to existing practice at Google where analysis writers tune checks based on results from running over the Google codebase [186].

Tailoring applicability and usability. Patterns may apply analysis-wide (e.g., **substr-model**) or work best at a local project level (e.g., **free-model** is unlikely to apply to all C programs). The particularity of patterns affects the size of false positive reduction. For example, the **create-socket** pattern removes only one false positive of 106 resource leaks reported by Infer. On the other hand, the size of false positive reduction is not the only significant metric. For example, even a single false positive report in `SpotBugs` due to **null-on-resources** breaks the continuous integration (CI) build of the projects we evaluated, and caused users to disable the check wholesale for their projects across all versions of Java. Because of this profound effect, bug severity and build integration must be weighed into analysis configuration mechanisms; our approach presents a new way to

cope with such complexities.

Analyzers on GitHub have numerous open issues related to "false positive" reports (`PHPStan` has 32 open issues, `Infer` has 36, and `SpotBugs` has 38), and our approach can fall outside the scope of these. For example, we may need type information to check whether a method or object may cause a false positive. As our approach is syntactic, resulting transformations do not guarantee, e.g., well-typed programs.

We note the compelling case for applying our technique in black box analysis settings. Users of commercial analyzers, like `CodeSonar`, do not have agency over the closed-source implementation or configuration options outside those provided by the distributor. Our approach demonstrates a new way to fine-tune results that is complementary to black box analysis.

True positive warnings in the modified program may appear at different lines compared to the original program. As a usability concern, affected lines in the modified program should map to those in the original. This is primarily an engineering concern, as transformations keep track of precise changes in offsets so that no information is lost.

Automated pattern inference. We note that inferring transformations from historical commits can reveal workarounds for false positives (as in the **const-strcmp** which we derived from the rsyslog commit). Although an interesting opportunity, we note that a number of our patterns perform changes that should only exist temporarily during analysis (**free-model**, **create-socket**, and **wrapped-resources**), and inference thus presents limited opportunity to learn such patterns. It is nevertheless an interesting avenue for future work.

3.5 Related Work

Program transformation has been used in various contexts to augment a procedure, technique, or system. Harman et al. introduce the idea of testability transformation [107, 108] where the goal is to transform a program to be more amenable to testing (e.g., by altering control

flow) while still satisfying a chosen test adequacy criterion. Program transformation can improve fuzz testing coverage and reveal more bugs [176] and enable new crash bucketing strategies to accurately triage bugs [200]. Failure-oblivious computing [184] adds (for example) bounds checking that allows programs to execute through memory errors at runtime. We similarly develop source-to-source transformations; however, we focus on improving analysis output fidelity. Our technique also aims to improve a static procedure and thus must be fast enough to integrate into static workflows. Randomized program transformation is an approach for testing static analyzers [77] and compiler internals, and excels at finding bugs in optimization passes [85, 213]. Our approach differs generally from these in using tailored program transformations to deterministically rewrite syntax.

Program transformation on intermediate representations (IR) can improve analysis precision (e.g., by adding bounds on arrays [52, 74, 131]). Recent work formalizes the impact of program transformations on static analysis in the abstract [162]; for example 3-address code transformation can introduce analyzer imprecision [144]. These works adopt a predominantly semantic view of program transformations and their influence on analysis; Cousot and Cousot develop a language-agnostic framework for reasoning about the correspondence of syntax and semantics under transformation [75]. These ideas underlie our intuition that semantic changes can improve analysis. However, abstract representations are difficult for developers to manipulate. Our work promotes changing program syntax directly as a proxy for inducing semantic changes that enhance analysis reasoning. In practice, programs are difficult to generically transform [156]. Recent advances for transforming multi-language syntax [199] however allows us to implement our approach in an efficient and declarative manner.

In practice analyzers compromise on soundness [142] and implementation tradeoffs manifest as implicit tool assumptions that are difficult to trace and modify [68]. Existing work shows that analyzer configuration options and suppression mechanisms fall short of developer needs in practice [67, 116]. Recent work by Gorogiannis et al. [99] emphasizes the

value of reducing false positives over false negatives, where the objective is to never report a false positive. In terms of this work, we introduce a new program transformational approach for nudging analyzers to report fewer false positives while sidestepping the difficulties of modifying analyzer implementations, assumptions, or configurations.

3.6 Summary

We introduced a new approach for effecting changes in static analysis behavior via program transformation. Our approach uses human-written templates that declaratively describe syntax transformations. Transformations are tailored to suppress spurious errors and false positives that arise due to problematic patterns and limitations in analyzer reasoning. We made the observation that analysis users have little agency over the format of analysis configuration options provided to them, but that program transformation offers a fresh primitive for leveraging influence over analysis behavior. To this end we showed that manipulating concrete syntax can resolve diverse and long-standing issues in existing analyzers, where configuration and suppression mechanisms fall short. Our evaluation presents the first study for empirically validating this program transformational technique, which we evaluated on active analyzers and large real-world programs.

At a high level, automated and selective program transformation presents a general way to influence analysis behavior, and is not restricted to static analyses. In the next chapter, we develop a new technique in a dynamic setting, this time demonstrating the capacity for automated program transformations to produce high quality bug reports by deduplicating crashing inputs generated through fuzz testing.

Chapter 4

Semantic Crash Bucketing

In this chapter we present Semantic Crash Bucketing, a generic method for precise crash bucketing via program transformation. In similar fashion to Chapter 3, we apply automated program transformation to tailor programs; now, however, to better deduplicate crashes produced by dynamic fuzz testing. Semantic Crash Bucketing maps crashing inputs to unique bugs as a function of changing a program (i.e., a semantic delta). The insight is to *approximate* real bug fixes with lightweight program transformation to obtain a precise classification of unique bug reports. Our evaluation shows that approximate fixes are competitive with using true fixes for crash bucketing, significantly outperforming built-in deduplication techniques for three state-of-the-art fuzzers.

4.1 Introduction

The advent of large scale fuzzing services, such as Google's OSS-Fuzz [23, 189] and Microsoft's fuzzing service [25], attest to the effectiveness of these automatic bug finding tools. When operating at scale, accurately identifying unique bugs is critical for (a) reducing time consuming manual debugging efforts [66, 178], (b) characterizing the effectiveness of automated bug-finding tools [42, 66, 157, 182, 209], and (c) ranking interesting crashing

test cases [66]. However, one outstanding challenge in effectively deploying automated fuzzing techniques is accurately identifying unique bugs during crash triage. Fuzzers often generate thousands of crashing inputs that ultimately correspond to the same bug [66], and the sheer number of crashing inputs preclude manual inspection. This is a hard problem, and an area of active research [76].

Automated crash triage techniques seek to approximately *bucket* multiple crashing (but ultimately equivalent) inputs [66, 76, 157, 178], to reduce the number of redundant bug reports an engineer must inspect by hand. At a high level, automated testing tools like fuzzers and symbolic executors typically use tool-specific, heuristic bucketing strategies. Both research and industry standard triage techniques have known limitations [76, 182]. Techniques may assume "best-effort" hardcoded values (e.g., the number of calls to consider in a call stack [18]) or require tool-specific instrumentation for feedback-driven approaches [21]. The varied sensitivity of such ad hoc techniques result in imprecise bug identification that can fail in two ways. *Overapproximation* occurs when multiple crashing tests caused by a single bug incorrectly bucket to more than one unique bug (i.e., duplicate bug reports). *Underapproximation* occurs when crashing tests due to multiple unique bugs are put in the same bucket [178, 209] (i.e., missed unique bugs).

Stack hash [157, 209] and branch sequence [16] techniques used in state-of-the-practice fuzzers [16, 18, 21] can suffer from both over- and underapproximation [178, 209]. Such techniques seek to determine bug uniqueness as a function of, e.g., crashing input [209], program traces [16, 76], program crash state [21], or a combination of these [66]. Recent (and more sophisticated) research advances propose to more precisely classify unique bugs using symbolic analysis [178], machine learning on crashing inputs [66], and backward taint analysis on program traces [76]. While such approaches promise more accurate bucketing by considering semantic program behavior (e.g., [76, 178]), their accuracy depends on sensitivity to a general semantic trait (e.g., symbolic branch uniqueness) and can still misbucket bugs. Built-in or hardcoded techniques further struggle to integrate specialist

knowledge that can produce more accurate output for classes of bugs.[1]

We present a new approach to identifying unique bugs in the context of fuzzing: by modifying the program itself. Our insight is that bugs can be characterized by a semantic transformation on the program under test. For example, patching one of two buffer overflows in a single execution can distinguish crashes uniquely caused by the second. Further, a fix can stop the same logical bug from manifesting on multiple unique execution paths.

Our insight draws on the fact that *fixing the program* offers a precise way to associate crashing inputs with a unique bug, since correct fixes should neutralize all crash-inducing inputs associated with the bug in question.

We introduce Semantic Crash Bucketing, which maps crashing inputs to bugs as a function of change (delta) in program semantics, where the delta approximates fixing the root cause of the bug. In general, root cause analysis is hard [128, 164], and automatically fixing bugs is an open problem [136]. However, existing work in automated program repair does demonstrate that programs can profitably be transformed to automatically improve quality [135, 145, 154]. The motivation behind our approach is that changing a program with *approximate* fixes can accurately and automatically constrain crashing behavior in a way that mimics real program fixes to detect unique bugs in fuzzer output.

Semantic Crash Bucketing contrasts with the usual sense of seeking program fixes with respect to a correctness validation oracle (such as tests [135, 145]). However, although the objective of Semantic Crash Bucketing is different from automatic program repair, it can similarly suffer from program transformations that overfit to the success criterion. For example, suppose a program contains more than one unique bug, each with independent fixes. Inserting `exit(0);` at the beginning of a program will satisfy the criterion of neutralizing all crashes, but will associate (and underapproximate) all unique bugs with a single fix. To be effective, program transformations must therefore have constrained semantic effects to precisely identify unique bugs under Semantic Crash Bucketing.

[1] https://twitter.com/azonenberg/status/966738179486134272

We propose a rule-based approach using fix templates to constrain the semantic transformations for crash bucketing. Our observation is that common bugs typically detected by fuzzers (e.g., buffer and integer overflows, null dereferences, etc.) have semantic properties that are amenable to a rule-based application of general fix templates (as found in analog APR work, e.g., [70, 122, 191]). At a high level, rule-based application of fix templates can integrate specialist knowledge of bug semantics into the triage process to produce more precise output. This attribute is analogous to targeted transformation for known static analysis issues, as covered in the previous chapter. We demonstrate Semantic Crash Bucketing for buffer overflows and null dereferences on real-world bugs in the CVE database [19]. Buffer overflows and null dereference vulnerabilities account for some of the most common software security weaknesses [139] and are frequently discovered through fuzzing [16, 182, 189]. Our contributions are as follows:

- **Semantic Crash Bucketing**, a novel technique to automatically identify unique bugs as a function of changing a program's semantics. Semantic Crash Bucketing groups crashing inputs by applying program transformations to the program under test. We use Semantic Crash Bucketing to identify imprecise crash reporting in fuzzers, and to compare the effectiveness of developer-written fixes and approximate fixes.
- **Approximate fixes.** We present an automated procedure using bug-fixing patch templates and rule-based application of patches to approximate correct fixes. In general, correctly and automatically fixing a program is hard. The key insight is that the effectiveness of approximate fixes is competitive with using correct fixes for identifying unique bugs. We instantiate Semantic Crash Bucketing with approximate fixes for real-world bugs commonly found by fuzzers: buffer overflows and null dereferences and demonstrate effectiveness.
- **Empirical evaluation**. We comparatively evaluate Semantic Crash Bucketing using developer-written fixes and approximate fixes with deduplication techniques of three

state-of-the-art fuzzers (AFL-Fuzz [16], CERT BFF [18], and Honggfuzz [21]). We show with Semantic Crash Bucketing that approximate fixes associate crashing inputs precisely with respect to ground truth (i.e., no under- or overapproximation) for 19 out of 21 bugs in 6 projects compared to ground truth fixes. We also show that bucketing with approximate fixes is more precise than built-in deduplication of all three fuzzers. Our results are available online.[2]

4.2 Motivating Example

```
1  --- a/src/select.c
2  +++ b/src/select.c
3  @@ -4153,7 +4153,7 @@ static int selectExpander(Walker *pWalker, Select *p){
4         /* A sub-query in the FROM clause of a SELECT */
5         assert( pSel!=0 );
6         assert( pFrom->pTab==0 );
7  -      sqlite3WalkSelect(pWalker, pSel);
8  +      if( sqlite3WalkSelect(pWalker, pSel) ) return WRC_Abort;
9         pFrom->pTab = pTab = sqlite3DbMallocZero(db, sizeof(Table));
10        if( pTab==0 ) return WRC_Abort;
```

(a) `SQLite`: a developer fix that avoids a null dereference.

```
1  --- a/src/resolve.c
2  +++ b/src/resolve.c
3  @@ -164,6 +164,9 @@ int sqlite3MatchSpanName(const char *zSpan, const char *
       zCol, const char *zTab, const char *zDb){
4     int n;
5
6  +  if(zSpan == NULL) {
7  +    exit(101);
8  +  }
9     for(n=0; ALWAYS(zSpan[n]) && zSpan[n]!='.'; n++){}
10    if( zDb && (sqlite3StrNICmp(zSpan, zDb, n)!=0 || zDb[n]!=0) ){
11      return 0;
```

(b) `SQLite`: autogenerated approximate fix for the null dereference.

Figure 4.1: Two fixes for a null dereference in `SQLite` 3.8.9. The actual fix is shown on top (commit `10c478e`). Our approach automatically generates the patch on the bottom.

[2] https://github.com/squaresLab/SemanticCrashBucketing

AFL-Fuzz is known to find null dereference and memory corruption bugs in even well-tested software [15]. Consider one such bug found in `SQLite`: a null dereference that was later fixed by the patch in Listing 4.1a. The `sqlite3WalkSelect` function (Line 7) walks the expression tree of a SQL select statement. The return value of `sqlite3WalkSelect` can indicate an error in a `SELECT ... FROM ...` statement, but the return value is not checked. This missing check can lead to a null dereference downstream during execution due to an invalid `FROM` clause. The fixing commit message says:

```
Make sure errors from the FROM clause of a SELECT cause analysis to
abort and unwind the stack before those errors have a chance to
mischief in the "*" column-name wildcard expander.
```

The developer fix thus checks the return value of `sqlite3WalkSelect` and aborts, avoiding any null dereferences downstream (Line 8, Figure 4.1a).

Current fuzzers and symbolic executors can find many different crashing inputs that trigger bugs like these. For example, slight modifications in a crash-inducing `SELECT...FROM...` input could follow a different sequence of calls or branches, but still trigger the same bug. Existing techniques use generic heuristics to identify unique crashes from a set of many generated inputs. Call stack hashes [18, 21, 42, 95, 157] are predominant; instrumentation-based fuzzers may use program execution paths sensitive to branch sequences [21]. These heuristics are fast and moderately effective, but remain imprecise, because they are sensitive to inputs that vary program execution in a way that is unrelated to the actual bug. Depending on the heuristic and inputs, fuzzers report many duplicate crashes as unique.

Our approach defines bug uniqueness in terms of *program transformation*. The motivation is that fixing a bug (as the developer did in Figure 4.1a) ideally "catches" all crashing inputs related to the bug, irrespective of call stacks or other program execution paths.[3] The challenge is that finding true fixes is hard. Automated root cause analysis is difficult

[3] And, under the correctness assumption of the fix, any other crashing input is associated with a different unique bug.

and expensive [128, 141], especially for bugs like this one, that requires deep reasoning.

Our primary insight is that simpler *approximate* fixes can substitute for real fixes to precisely bucket crashing inputs (i.e., as the original fix would). For example, Figure 4.1b presents an autogenerated approximate fix for the same `SQLite` null dereference bug using our approach. Semantically, it safely aborts the program if `zSpan` is `null`. It turns out that the `SQLite` bug leads directly to `zSpan` being `null` at this later program point (i.e., when the input statement contains a * expander described in the commit message). The approximate patch "catches" all crashing input variants (i.e., generated by a fuzzer) with the same precision as the actual patch. Throughout the rest of this chapter, we use the term "precise" in this sense: to refer to a correct bucketing with respect to a ground truth developer fix for discriminating crashes. I.e., we do not define a fundamental semantic trait that such ground truth fixes must have, but consider various traits of fixes for different bug classes.

Our approach uses syntactic templates and configurable "semantic cues" to generate approximate patches to mimic developer fixes. Syntactic templates are implemented using `Comby`, as described in Chapter 2. Semantic cues act as predicates for applying patch templates, which fire based on syntactic patterns and dynamic runtime behavior. A concrete example is "Check whether any dereferenced variables at program point p is `null`. If so, return the variable name". A patch template can then be instantiated with the specific variable at an appropriate program location. In general, templates and rules for patch generation and application are specified just once per bug class (e.g., null dereferences and overflows.). We describe the procedure fully in Section 4.4, but provide a brief summary here for null dereferences. The patch template for null dereferences checks whether a variable is `null`, and safely aborts the program if so. The rewrite template contains a "hole" for the variable to check, and must be instantiated with a concrete variable. We configure the procedure to check for a semantic cue: whether variables are null at the point of crash using a debugger environment. In this case, our procedure finds that `zSpan[n]` could be a

problematic dereference, and dynamically checks whether `zSpan` is null when the program crashes. Variable `zSpan` is found to indeed be `null`, generating the patch in Figure 4.1b. The patch is validated to confirm that the modified program no longer crashes for the input. That is, the autogenerated patch approximates the real fix effectively because it discovers and fixes the related null dereference triggered downstream during execution, even though it does not deeply address the root cause.

In essence, applying lightweight program transformation reduces noise compared to typical deduplication heuristics by focusing on the semantic properties of the bug. At the expense of slight up-front cost per bug class, our approach provides a configurable mechanism that is sensitive to the semantic property of the bug to more precisely identify uniqueness.

A configurable approach is important because bugs exhibit different semantic traits to which program transformation must be sensitive. For example, null dereferences cause an immediate program crash which allows us to identify possible causes at the point of crash. On the other hand, buffer overflows typically only cause a crash once corrupted memory is accessed, and not when the overwrite actually occurs. Handling overflows therefore requires a different strategy (see Section 4.4).

Fuzzers can also underreport unique bugs. For example, under a naïve call stack approach, two unique null dereferences in a single function will be reported as just one unique bug. Our technique can identify each bug uniquely via independent program transformations.

4.3 Semantic Crash Bucketing

Semantic Crash Bucketing is a general method for bucketing crashes *in terms of program transformation* (i.e., a semantic delta). Semantic Crash Bucketing can be performed with arbitrary program transformations. Our goal in this section is to develop a way for

determining how well approximate fixes identify unique bugs compared to (a) ground truth fixes and (b) existing methods in fuzzers. We now introduce the problem definition and application of SCB for detecting inaccurate error reports.

4.3.1 Problem Formulation

A *bug* in this context is a software flaw that leads to an error (i.e., undesirable program behavior); an error is a deviation from expected behavior defined by a validation oracle. We focus on the types of bugs typically found by fuzzers, namely those that induce runtime crashes. For such bugs, the error oracle is signaled by a runtime failure: a crash results in `SEGFAULT`.

Semantic Crash Bucketing groups crashing inputs according to a program change that nullifies those inputs (i.e., cause the inputs to no longer crash the program). Thus, a true fix for a unique bug maps all crashing inputs for that bug to a unique bucket. Grouping crashes as a function of known fixing patches is a de facto method for establishing ground truth classification of fuzzer crash reports [66, 124]. We use this idea to develop a general method of identifying misbucketing (e.g., duplicate crash reports) arising from approximate fixes and fuzzers.

Ideal bucketing. We begin by defining an IDEAL BUCKETING, where the correct fixing patches for unique bugs in a program are known or presumed. In this definition we qualify bug uniqueness fully as a function of the known fix. I.e., a single fixing patch implies a single unique bug with respect to the set of crashing inputs it fixes (irrespective of how much code is changed). It follows that this definition can represent ground truth to measure the effectiveness of our approach, given a set of bug-fixing patches that we assume are correct (Section 4.5). The intuition is straightforward: some known or presumed program transformation T_i fixes all crashing input associated with a bug i, and only those inputs. T_i is by construction the theoretically ideal oracle transformation that correctly fixes the bug

i and thus all of the crashing behavior it can cause. In practice, we may think of such a transformation as a correct developer-written patch for a single bug.

We express IDEAL BUCKETING in terms of unique bugs, the crashes they induce, and their fixes. Let $i \in n$ be the identifier for a unique bug i of n unique bugs in a program P. A unique bug i is associated with a set of one or more crashing inputs, which we denote by a bucket b_i. Let $T_i : P \to P$ be a function that applies a correct fix to the program P, for unique bug i. A correct fix T_i fixes all crash-inducing inputs b_i due to i, but none of the crashes due to a different bug j with crashing inputs b_j.

We express all buckets containing crashing inputs uniquely fixed by known T_i, $i \in n$ as disjoint partitions $B = b_1 \uplus \cdots \uplus b_n$ under the correctness assumption of T_i. For a particular T_i, IDEAL BUCKETING implies:

$$
\begin{aligned}
&\forall\, b_i \in B, \\
&\forall\, b_j \in B \setminus b_i \ \texttt{s.t.} \\
&\quad \forall\, c_i \in b_i, \langle T_i(P),\ c_i \rangle \not\rightsquigarrow \mathit{crash} \\
&\quad \forall\, c_j \in b_j, \langle T_i(P),\ c_j \rangle \rightsquigarrow \mathit{crash}
\end{aligned}
$$

Where $\langle T_i(P), c \rangle \not\rightsquigarrow \mathit{crash}$ expresses that the program P under transformation of fix T_i and executed on crashing input c does not induce a runtime crash. IDEAL BUCKETING for a bug i expresses that the fix T_i associates non-crashing behavior with all previously crashing inputs $c \in b_i$, but not any crashes for other buckets $b_j \in B \setminus b_i$.[4]

One subtlety of IDEAL BUCKETING is the special case where a single input may trigger multiple bugs. For example, two separate buffer overflow copies (i.e., two bugs b_1 and b_2) along the same execution path may overwrite the stack (twice) in a single execution. From our definition, neither corresponding fix T_1 nor T_2 will bucket the crashing input. However, we can extend the definition to account for *composition* of transformations T_1

[4]Note: if $B \setminus b_i = \varnothing$ then the constraint on b_j holds vacuously.

and T_2 to place such a crashing input into a separate bucket that represents a composite fault. Although conceptually useful, we focus on logically discrete fixes (based on developer patches) to associate crashing inputs with bugs so that it is tenable to experimentally compare real and approximate fixes. In practice, fuzzer-generated inputs typically trigger single bugs, and our results corroborate this observation. Classifying composite faults is an open problem [98] and we leave the consideration of using program transformation for classifying such faults to future work.

4.3.2 Detecting Duplicates

Our goal in fuzz triaging is to approximate the ground truth IDEAL BUCKETING strategy, minimizing overhead and confusion for the engineer using a fuzzer to identify defects. Current techniques approximate by, e.g., unique call stack hashes or unique branch sequences. When such techniques fail, *misbucketing* occurs. Misbucketing can be classified into two categories [178]:

1. duplicate bug reporting and
2. suppressed unique bugs: unreported unique bugs that are missed by crash bucketing (or "over-condensing").

In this dissertation, we address the first case of duplicate bug reports. We now describe how we detect duplicate bug reports in terms of fixing transformations T_i, where IDEAL BUCKETING does not hold. Consider two example bug reports produced by a fuzzer: bug 1 with a crash bucket $b_1 = \{c_1\}$ containing crashing input c_1, and bug 2 with $b_2 = \{c_2\}$. We say that b_2 is a *duplicate bug report* if c_2 actually crashes the program due to bug 1. That is, the correct bucketing implies $b_1 = \{c_1, c_2\}$ and no bug 2 should be reported. Duplicate misbucketing wrongly implies bug uniqueness, increasing the triage burden of engineers processing fuzzer output.

In an *imprecise* bucketing B, duplication occurs when the following is true for a particular

T_i:

$$\exists\ b_i \in B,$$
$$\exists\ b_j \in B \setminus b_i \texttt{ s.t.}$$
$$\forall\ c \in b_i, \langle T_i(P),\ c\rangle \not\rightsquigarrow \textit{crash}$$
$$\exists\ c_{\text{dup}} \in b_j, \langle T_i(P),\ c_{\text{dup}}\rangle \not\rightsquigarrow \textit{crash}$$

That is, some crash $c_{\text{dup}} \in b_j$ *actually fixed* by T_i is considered a crash for a different unique bug j, belonging to b_j. By our correctness assumption of T_i, any crash fixed by T_i must belong to b_i for IDEAL BUCKETING to hold. Note that if c is the only crash in b_j then a unique bucket is implied, resulting in a duplicate report of bug i as some other bug j that should not exist.

In summary, given correct T_i's, we can determine ground truth IDEAL BUCKETING and detect duplicate bug reports as deviations from IDEAL BUCKETING.

4.3.3 Semantic Crash Bucketing Procedure

Our formulation leads to a straightforward procedure for identifying misbucketing in fuzzers. Figure 4.2 illustrates the process. A `Fuzzer` takes a program P and input to generate a set of crashes $C = c_1, \ldots, c_n$ ❶. The fuzzer reports a set of crashing input according to its built-in method for identifying unique bugs. We represent the fuzzer output as a disjoint set of unique bugs indexed by I: $B_{\textit{fuzzer}} = \biguplus_{i \in I} b_i$ ❷.

As a matter of practicality, a fuzzer does not, by default, preserve all generated crashing inputs. Instead, a fuzzer discards any crashing input it believes triggers a bug it has already seen, and typically outputs one representative crash for each bug/bucket it considers unique. This is expressed as $B_{\textit{fuzzer}} = (b_1 = \{c_1\}) \uplus (b_2 = \{c_2\}) \uplus \cdots \uplus (b_n = \{c_n\})$.

The function SCB takes as input the set of crashes C and a ground truth fix T_j ❸. For

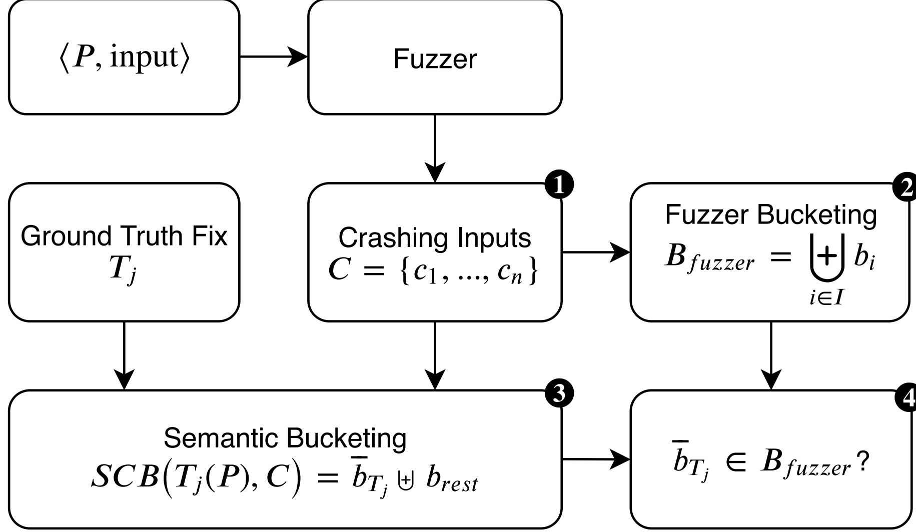

Figure 4.2: The Semantic Crash Bucketing procedure.

a single fix T_j, SCB partitions the set of crashes C into a disjoint set $\bar{b}_{T_j} \uplus b_{rest}$ by running each crash $c \in C$ on the modified program $T_j(P)$. The set $\bar{b}_j$ contains all inputs fixed by T_j, as in the Ideal Bucketing case, while b_{rest} contains all inputs that still cause $T_j(P)$ to crash. The final step ❹ tests if the crashes in a nonempty bucket $\bar{b}_{T_j}$ distinguished by T_j is contained in B_{fuzzer}. Because B_{fuzzer} contains a partition of unique bugs with just one representative crash in any b_i, there are only two outcomes for the test $\bar{b}_{T_j} \in B_{fuzzer}$: (1) $|\bar{b}_{T_j}| = 1$ and is equal to some b_i in B_{fuzzer}, implying that B_{fuzzer} precisely buckets the crashing input for a bug j, or (2) $|\bar{b}_{T_j}| > 1$ implying that the crashes in $c \in \bar{b}_{T_j}$ are partitioned across multiple buckets in B_{fuzzer}, implying that the fuzzer reported duplicate bugs. For simplicity, Figure 4.2 illustrates the procedure for a single fix T_j that fixes at least one crashing input in C. Ideal Bucketing checks that every crashing input in C can be fixed (and bucketed) uniquely by one or more corresponding fixes T.

4.4 Generating Approximate Fixes

This section explains how we instantiate our approach to perform SCB using *approximate fixes*. In practice, a developer fix provides the best assurance of correctly fixing a known bug, which we accept as ground truth T for SCB. However, our goal is to reduce the burden on developers to triage fuzzing output when the crash's fix is not immediately known. In general, fixing arbitrary bugs automatically is hard [136]. Our core insight is that an approximate fix $\widehat{T}$ is competitive with using T to identify unique bugs under SCB. In our approach, $\widehat{T}$ is an *automatic* production encoding the semantic properties necessary to fix possible crash-inducing bugs. To demonstrate, we instantiate SCB with approximate fixes on null dereferences and buffer overflows in C programs.

4.4.1 $\widehat{T}$ Production.

At a high level, $\widehat{T}$ is a production of a function $G(P, \mathcal{T})$ that takes two inputs: the source program P and a crash trace $\mathcal{T}$. A crash trace is produced by executing P on a single crashing input c. G generates patches from fixing templates, and applies them to the source. Patch application is predicated on certain syntactic patterns in the program source and behavior recorded in a dynamic trace. We refer to these predicates as *semantic cues* that are sensitive to semantic properties of a bug class. If the predicates are not satisfied, the program is not modified.

We concretely represent $\widehat{T}$ as a source-level patch. This has two advantages. First, patches can be used as better bug reports [204], supporting human triage and debugging. Second, patches can apply without actually running the program, meaning static analyses (e.g., static symbolic execution) can also benefit from SCB.

We use `GDB` and `ltrace` to obtain dynamic crash traces. In principle, any dynamic technique or analysis can enrich the space of semantic cues to trigger program modification. We now describe in concrete terms how we obtain $\widehat{T}$ for null dereferences and buffer

overflows.

4.4.2 Null Dereferences

Null dereferences are typically fixed in one of two ways: correctly initializing a variable or checking whether a variable is null before dereferencing it [191, 211]. At a semantic level, a fix must enforce a nonnull property for a variable that results in a null dereference crash. We use the rewrite template in Figure 4.3 to approximate fixing a null dereference. `:[pvar]` is a hole (as in Section 2) substituted with the offending program variable.

```
if (:[pvar] == null) {
  exit(101);
}
```

Figure 4.3: A template for null dereferences

The patch approximates error handling by exiting the program on condition of `:[pvar]` being `null` (similar to the common C idiom of `return -1;`). While simply exiting appears overly simplistic, it is in fact appropriate for our objective to accurately bucket crashing input. Consider if we chose a different strategy by returning a value or initializing `:[pvar]`. Besides the difficulty of correctly inferring appropriate values, we risk the possibility that the modified program may continue executing and cascade errors or crash in other unexpected ways. Without complete information of the root cause to actually fix the bug, exiting is a conservative strategy: it acts as an assertion ensuring the desired nonnull semantic property. The correct fix in our motivating example supports this strategy: `SQLite` conservatively aborts for error cases (but does some extra work propagating the error up the call stack). Since the template can be changed, our method does not preclude other possibilities. However, our experiments show that the template in Figure 4.3 approximates true fixes well enough for precise crash bucketing.

The rewrite template definition is only part of the larger problem: generating the final

Match Template	Extract Template
`:[1]->:[2]`	`:[1]`
`:[1]->:[2]->:[3]`	`:[1]->:[2]`
`:[1]->:[2][:[3]]`	`:[1]`
`:[1][:[2]]`	`:[1]`
`*:[1]`	`:[1]`
`(:[1])->:[2]`	`:[1]`

Table 4.1: We use `Comby's` rewrite capability to identify pointer dereferences on variables, and extract those variables. The **Match Template** matches syntactic dereference patterns, and the **Extract Template** extracts the program variable or expression that is dereferenced. Ultimately, we are interested in checking whether the extracted variable is null at the point of the crash.

patch $\widehat{T}$ also relies on identifying the appropriate program variable and location to insert the patch. Semantic cues from a `GDB` trace inform patch application: whether a variable dereference at the point of crash is null. For example, our approximate patch in Figure 4.1b checks the variable `zSpan`. The general procedure for finding such crash-inducing variables works as follows:

1. Attach `GDB` to the program, run it on the crashing input.
2. Extract the source line and code reported at the crash.
3. Parse the code for pointer dereference syntax (e.g., `p->q`).
4. Extract program variables that are dereferenced (e.g., extract `p` from `p->q`). We notably use `Comby`'s rewriting capability to parse and extract these syntactic patterns (see Table 4.1). We then test, using `GDB`, whether an extracted variable is `null` in the debugger environment. We do this for each possible extracted variable found by the patterns in Table 4.1.
5. For each variable that is null, return the variable and associated line number. If not, move backwards a basic block and continue from (3).

If the above procedure succeeds, we substitute the template program variable and insert the candidate patch at the source location immediately before the null dereference. The null

check could possibly be placed earlier, and a true fix may indirectly prohibit a particular variable from being null (cf. the correct `SQLite` fix in Figure 4.1a). Our decision is an inexpensive compromise that we show works well in practice.

Before we use the patch for SCB, we first validate that the modified program no longer crashes for input c. If so, we have found a crash-fixing patch. The patch generation procedure can produce more than one candidate patch, but our implementation takes the first crash-fixing patch for bucketing.

Compared to using `Comby`, there are alternative ways to inspect whether a variable is a null pointer on crashing input. In practice, we found that using only `GDB` was unreliable: we could not easily extract whether a variable is indeed a null pointer, versus just a constant with the value zero. Inspecting the corresponding syntax thus provides a robust alternative. Initially we developed a `Clang` plugin to extract variable and type information. This solution was brittle: `Clang` may fail to parse certain files (e.g., it would fail on GCC-compiled code containing inline assembly). We found `Comby` to provide a robust and simple solution using lightweight syntactic patterns. We iteratively developed the six patterns in Table 4.1 during our experiments until they covered all bug-inducing dereference syntax. Interestingly, we reached a fixpoint of six patterns for 18 unique null dereference bugs.

4.4.3 Buffer Overflows

Buffer overflows are a class of memory corruption bugs commonly discovered by fuzzers [16, 189]. Buffer overflows are typically fixed by performing array bounds checking on memory accesses. Our approximation to fixing buffer overflows thus focuses on array length as the underlying semantic property to change. Inferring array bounds can directly assist suggesting approximate fixes for arbitrary overflow bugs, but generally requires additional analysis techniques and remains an open problem [91, 96, 105]. Our approach is to truncate memory writes that may cause invalid accesses. Applications in failure-oblivious computing [184]

```
// Modify a possible overflowing memcpy call
size_t angelic_length = 1;
memcpy(:[dst], :[src], angelic_length);
```

Figure 4.4: A template for `memcpy`. `:[dst]` binds to the destination argument for the original `memcpy` call, and `:[src]` is binds to the source argument.

and exploit mitigation [141] use a similar mechanism.

Unsafe C library functions commonly trigger buffer overflows [91, 130, 141, 160] and persist in modern software.[5] Our approach applies templates for common C library functions, such as `memcpy`, `strcpy`, `sprintf`, `gets`, `strcat`, etc.

We give an example template for `memcpy` in Figure 4.4; the templates for other overflows are conceptually similar. We rewrite existing calls and restrict the length of data copied to a default concrete value of 1. Restricting data to only one byte approximates a conservative *angelic value* [63] that is likely to lead to non-crashing program termination. Note that other possibilities exist. We may, for example, instrument the code to obtain actual angelic values observed at runtime and use these to construct fixes. Our experiments show that our current choice works well for precise bucketing of the bugs we consider.

Compared to approximating null fixes, overflow fixes do not attempt to stop execution: placing a condition on the length of a potential buffer proves problematic if we do not know its bounds. Conversely, simply exiting before calling an unsafe function will overfit to unique crashing inputs that would crash after the function call. In addition, while memory corruption occurs *during* execution of the C library functions, the program only *crashes at a later point*: once an invalid memory access occurs in the heap, or when a corrupted return address is accessed on the stack.[6] These behaviors motivate different semantic cues compared to null dereferences, and emphasize the importance of a configurable approach. For buffer overflows, we implement a procedure to discover possibly problematic library

[5] A `strcpy` vulnerability has been found in the Linux distribution as recently as 2017 [22].

[6] Memory fence-posts can detect overwrites immediately, and don't require a program to `SEGFAULT`. This requires code instrumentation and extra shadow memory that hurts fuzzing performance. Approximate fixes can be adapted accordingly, but we currently do not assume such instrumentation.

calls and resolve their location. A patch template like Figure 4.4 then replaces the call. The steps are as follows:

1. Use `ltrace` to obtain a trace of library calls from the crashing program run.
2. Working backwards, resolve the source location of library calls in the trace for which we have fixing templates.
3. Apply the template at the location and rerun the program on the original crashing input.
4. If the program no longer crashes, emit the approximate fixing patch $\widehat{T}$. Else continue from step (2).

Similar to null dereferences, we validate that the program no longer crashes for any change done in step 3, and use the first crash-fixing patch for bucketing.

Extending Semantic Crash Bucketing. The patch templates and rules for patching are embedded in Python scripts and are easy to change. Users can define their own patch templates and semantic cues for patch application depending on the semantic properties of the bug types or application-specific APIs. The `GDB` interface and `ltrace` output is available in the scripting framework for customization. Additional analysis tools can be integrated (e.g., `valgrind`), though naturally this requires some extra effort.

4.5 Experimental Design

Ultimately, we want to know how well approximate fixes $\widehat{T}$ distinguish unique crashes compared to (a) ground truth bucketing by T (developer fixes) and (b) built-in fuzzer deduplication (the previous state-of-the-art). We conduct a controlled experiment with real bugs for which we know the ground truth fix (Section 4.5.1). Unfortunately, for the purposes of our experiments, state-of-the-art fuzzers do not all neatly decouple fuzzing campaigns from crash deduplication (e.g., deduplication is invoked during fuzzing iterations).

Instead, we first generate, for each bug, an upper bound of inputs that trigger the same bug (i.e., a "crash corpus") which aim to exercise different execution paths triggering the same bug (Section 4.5.2). We then provide this crash corpus as input to each fuzzer, and run a campaign for a fixed length (2 hours), forcing the fuzzer to perform deduplication on the crash corpus during fuzzing iterations (Section 4.5.3). We use the developer fix and apply SCB to obtain the ground truth number of duplicate bug reports *after* the campaign (which includes each fuzzer's deduplication effort on the corpus). We then apply SCB with *approximate fixes* and measure (a) the difference from ground truth, and (b) deduplication improvement over existing fuzzers.

Hardware. We ran our experiments on an Ubuntu 16.04 LTS server with 2 Xeon E5-2699 CPUs and 20GB of RAM. Crash Corpus generation and fuzzing campaigns all ran on a single CPU core. We used four cores to recompile when validating whether an approximate fix stops a crash.

4.5.1 Bugs with Ground Truth

We evaluate on a sample of 18 null dereference and 3 buffer overflow bugs in 6 real-world projects. For each bug we (a) extracted a ground truth developer fix from the project and (b) sourced a crashing input that triggers the bug (e.g., from online bug reports). We extracted fixes by manually inspecting individual commits and commit messages that indicated a bug fix, and performed delta debugging [215] to confirm that a commit fixed a crashing input. After confirming whether an extracted fix nullifies the initial crashing input, the fix was accepted as ground truth.

Projects with multiple bugs. `SQLite` is well-tested, popular database software; `w3m` is a text-based web browser. For these projects, we curated datasets of multiple bugs in a single revision. To be useful, a deduplication strategy should correctly bucket crashing inputs associated with a bug, but *only* that bug (and not those for other bugs). That is, $\widehat{T}$

should be as close to IDEAL BUCKETING as possible, giving strong assurance that $\widehat{T}$ does not overfit the input crashes.

Thus, we curated a dataset of fixes for 12 null dereference bugs in a *single* SQLite revision. This is an onerous task because developer fixes are often interspersed over long periods of time[7] and fixing patches cannot always be automatically applied to previous revisions due to intermediate code changes. In addition, a single patch may contain multiple fixes, which we must separate for each respective bug. We therefore manually minimized and backported patches to support a large, controlled ground truth study on multiple bugs in SQLite.

We selected w3m out of a list of projects with reported CVEs [24] and found that it also has multiple null dereferences for which we could find developer fixes and crashing inputs that work on the same revision.[8] We demonstrate SCB on four null dereference bugs on a single revision of w3m.

Other projects. We identify two null dereference bugs in different versions of PHP, a large, popular project with well-documented bugs and ground truth patches. We demonstrate SCB on real-world overflow bugs in R, a large and popular software suite for statistical computing; Conntrackd, a networking utility; and libmad, an MPEG audio decoder. We apply SCB to a strcpy vulnerability in R.

As an interesting demonstration of real-world utility, we applied SCB to two 0-days that we discovered during prior recreational fuzzing campaigns (a strcpy vulnerability in Conntrackd and a memcpy vulnerability in libmad). That is, we applied SCB *before* a developer fix was available. Roughly 10 months later, the developers supplied us with correct fixes. Both fixes bucket precisely the same crashing inputs as our automated approximate fixes in the experiments.

[7]The SQLite bugs were fixed over a period of four months.

[8]https://github.com/tats/w3m/issues?q=Null+pointer+is:closed

4.5.2 Crash Corpus Generation

For each bug, we generate a large baseline corpus of crashing inputs from the initial crashing input, aiming to exercise different execution paths triggering the same bug. We use this corpus to test how well each fuzzer's deduplication method copes with varying behavior that trigger the same bug. Although a typical fuzzing campaign begins with one or more non-crashing seed files as input, it is hard to trigger a specific bug starting with arbitrary seed files: the input search space is huge, and fuzzing nondeterminism means it is difficult target specific areas of code. Isolating features in test cases is one strategy for producing crashing test cases that may correspond to the same bug [40, 102], but can take several days to produce a large test set. Instead, we pursue a conceptually similar approach, mutating an initially crashing input to explore different execution paths that trigger a particular bug. We then use this corpus as input to the other state-of-the-art fuzzers.

To do this, we use the existing "Crash Mode" procedure implemented in AFL-Fuzz [16]. The crash exploration procedure tracks branches executed by the input, and mutates input to try and force execution along different branches, where the objective function is to preserve crashing behavior. Inputs that fail to explore interesting paths or crash the program are discarded. We run crash exploration for two hours *per bug*, producing crash corpora of related inputs for each bug's crashing seed file.

4.5.3 Evaluating Fuzzers

We compare to three state-of-the-art fuzzers: AFL-Fuzz [16], CERT BFF [18], and Honggfuzz [21]. These fuzzers are frequently used in industrial and research settings [17, 189, 209] and implement different deduplication techniques. In general, fuzzers do not decouple fuzzing campaigns from crash deduplication; crashes are deduplicated during fuzzer iterations. To trigger crash deduplication, we seed fuzzing campaigns for each fuzzer with the crash corpus.

Industrial-strength fuzzers are highly configurable. We sought to evaluate on default options across varying parameters in coverage-based fuzzing, call stack depth, branch sequences, and point-of-failure information. We evaluate on five configurations across the three fuzzers:

AFL-Fuzz. We use AFL-Fuzz in its default configuration. AFL is instrumentation-driven, and keeps track of branches taken during fuzzing. Roughly, this means that AFL is sensitive to uniquely executed paths. AFL's default method for fuzzing uses the same mechanism as "Crash Mode", starting from a non-crashing seed and with an objective of discovering arbitrary crashes. One key difference, however, is that "Crash Mode" does not deduplicate the crash corpus by default. Therefore, to approximate AFL's deduplication in a real campaign (while avoiding a redundant fuzzing campaign), we use AFL's own minimization procedure directly on the crash corpus, then remove equivalent duplicates.

CERT BFF. We run CERT BFF in its default configuration, which uses a call stack hash based on, by default, the last *five* calls (frames) leading to a crash. This number is configurable. Thus, for the second configuration, we set BFF to use a call stack of just *one* frame to determine bug uniqueness. BFF also invokes a built-in input minimizer while fuzzing on-the-fly.

Honggfuzz. We run Honggfuzz in its default configuration, which uses a call stack hash of seven calls. By default, Honggfuzz considers information at the point of failure when a crash occurs (e.g., the last known PC instruction and faulting address) to report uniqueness. Honggfuzz can enable a feedback-driven fuzzing mode, provided a program is compiled with the coverage instrumentation. In the first configuration, we disable coverage; in the second, we enable coverage.

Note that due to input mutation during the campaign, a fuzzer may trigger a bug that we do not have a fix for. As a final post processing step, we use the ground truth fix T to filter out only the crashes fixed by T.

Table 4.2: Semantic Crash Bucketing results. For each fuzzer configuration, we show the Ground Truth number of duplicate crashes reported by the fuzzer (GT) compared to the number of duplicate crashes reported using approximate fixes with SCB (SCB+$\widehat{T}$). **Crash Corpus** is the number of crashing inputs that initially seed fuzzing campaigns for each configuration. For example, Bug #1 (first row) in the HFuzz, GT column shows that HFuzz reports 10 duplicates (determined by the ground truth fix), while the approximate fix (SCB+$\widehat{T}$) reports 0 duplicates. When SCB+$\widehat{T}$ reports 0 duplicates, it is as precise as ground truth.

Project	**Type**	**ID**	**Crash Corpus**	**AFL**		**BFF-5**		**BFF-1**		**HFuzz**		**HFuzz-Cov**	
				GT	SCB+$\widehat{T}$	GT	SCB+$\widehat{T}$	GT	SCB+$\widehat{T}$	GT	SCB+$\widehat{T}$	GT	SCB+$\widehat{T}$
SQLite	Null deref	1	191	25	0	2	1	1	1	10	0	9	1
		2	482	85	0	2	0	1	0	4	0	2	0
		3	153	38	0	6	0	0	0	16	0	14	0
		4	326	48	0	0	0	0	0	1	0	0	0
		5	139	34	0	0	0	0	0	0	0	0	0
		6	66	21	0	0	0	0	0	0	0	0	0
		7	97	20	0	0	0	0	0	0	0	0	0
		8	235	82	0	1	0	0	0	3	0	3	0
		9	389	29	0	1	0	0	0	1	0	1	0
		10	270	65	0	0	0	0	0	1	0	1	0
		11	167	45	1	0	0	0	0	4	2	1	1
		12	108	36	0	0	0	0	0	0	0	0	0
		Subtotal	**2,623**	**528**	**1**	**12**	**1**	**2**	**1**	**40**	**2**	**31**	**2**
w3m	Null deref	13	458	103	0	25	0	1	0	75	0	77	0
		14	545	23	0	0	0	0	0	0	0	0	0
		15	507	36	0	0	0	1	0	6	0	4	0
		16	525	11	0	0	0	1	0	0	0	0	0
		Subtotal	**2,035**	**173**	**0**	**25**	**0**	**3**	**0**	**81**	**0**	**81**	**0**
PHP	Null deref	17	81	8	0	0	0	0	0	0	0	0	0
		18	272	32	0	0	0	0	0	0	0	0	0
R	Overflow	19	7	5	0	3	0	0	0	145	0	198	0
Conntrackd	Overflow	20	25	0	0	0	0	0	0	770	0	427	0
libmad	Overflow	21	138	8	0	1	0	0	0	1	0	0	0
		Total	**5,181**	**754**	**1**	**41**	**1**	**5**	**1**	**1,037**	**2**	**737**	**2**

4.6 Experimental Results

Our main result is that SCB with approximate fixes is *just as precise* as using the ground truth fix for 19 out of 21 bugs across all configurations. Approximate fixes suffer only small imprecision, and perform significantly better deduplication compared to state-of-the-art fuzzer deduplication in our experiments.

Speed and project size. Automatic patch generation for approximate fixes is fast. Generating a patch from crashing input *and* validating that it fixes the crash (including project recompilation) takes just 18 seconds on average across all bugs. The minimum time for patch generation and validation is 2 seconds, the maximum 49 seconds. Our sample uses large real-world projects, ranging from 12KLOC to 1MLOC.

4.6.1 Overall Results

Table 4.2 shows results. Each row corresponds to a unique bug, with assigned "ID". "Crash Corpus" is the number of crashing inputs that initially seed the fuzzing campaigns. We deduplicate crashing inputs for each bug using five fuzz campaign configurations (Section 4.5): **AFL**, **BFF-5** and **HFuzz** are default configurations for the three fuzzers. **BFF-1** configures BFF to use just one call in its call stack hash; **HFuzz-Cov** turns on coverage instrumentation for feedback fuzzing in HonggFuzz. "GT" is the Ground Truth number of duplicate reports for each respective configuration, which we obtain using the *actual* developer fix T for each bug. Column "SCB+$\widehat{T}$" is the number of duplicate bugs for a campaign reported using approximate fixes with SCB.

Except for Bugs 1 and 11 in `SQLite` (discussed subsequently), approximate fixes are *as precise* as the ground truth fix across all configurations. That is, approximate fixes detect and remove all duplicates across all fuzzing configurations for 19 out of 21 bugs. For projects `SQLite` and `w3m` containing multiple bugs, none of our approximate patches suppress any other unique bug. In aggregate, SCB with approximate fixes significantly

reduces the number of duplicate crash reports compared to the default configurations: from 754 and 1,037 to just two duplicates for **AFL** and **HFuzz**, respectively, and a reduction of 41 duplicates to one duplicate for **BFF-5**. In practice, crash reports produced by fuzzers must be further triaged manually. Our results show that applying approximate fixes can automatically cut down on the time that an engineer spends on further triage.

Ground truth fixes expose different "semantic sensitivities" of error reporting across configurations and bug types. AFL-Fuzz on average reports more duplicate bugs; this is expected due to its sensitivity to unique execution paths, especially for null dereferences. On the other hand, AFL and BFF report moderate numbers of duplicates for overflow bugs, whereas Honggfuzz reports hundreds of crashes for two stack-based overflows (Bugs 19 and 20). Honggfuzz's default sometimes considers portions of overflowing stack data to signal unique bugs. Bug 21 is a heap-based overflow, and does not adversely affect Honggfuzz compared to stack-based bugs. **BFF-1** uses just one call to calculate a unique stack hash per bug, and reports the fewest duplicate bugs. Although **BFF-1** appears to perform well, the configuration is nonstandard in practice because it has the caveat that unique bugs triggered in the same function are easily missed. None of the bugs in our sample exposes this weakness in the **BFF-1** configuration, but the weakness is common in real fuzzing campaigns. We include it as one extreme example where coarse, aggressive deduplication can be performed at the cost of potentially missing unique bugs.

Note that even for cases where a fuzzing configuration reports no duplicates for a particular bug, approximate fixes do *as well* as fuzzer deduplication, and *strictly better* for the majority of cases where duplicates are reported. This emphasizes an important point: approximate fixes uniformly bucket crashes via configurable sensitivity to bug-class semantics. Our results show that lightweight program transformation can effectively avoid imprecision due to varying (yet broadly applied) choices made by built-in fuzzer deduplication methods.

4.6.2 Project-specific Results

SQLite. Approximate fixes for `SQLite` perform identically to ground truth except for Bugs 1 and 11. Patches for Bugs 1 and 11 fail to fix 7 crashing inputs out of a larger duplicate crash set of 62 crashes reported by fuzzers. We analyzed these inputs and found that they generally trigger different crashing behavior downstream in execution that our approximate patches do not catch (but which correct patches handle earlier upstream). The implication is not severe: SCB+$\widehat{T}$ only reports 7 duplicates over all configurations, which is comparatively low compared to duplicate fuzzer reports.

w3m. SCB+$\widehat{T}$ perfectly simulates ground truth bucketing for `w3m`. Our approximate fixes are semantically close to developer fixes: each approximate patch checks the same program variable for `NULL` as the corresponding developer patch. Interestingly, Bug 13 produces far more duplicate crashes compared to the other three bugs across all configurations. This demonstrates a latent benefit of our approach: SCB can reveal properties about buggy behavior (e.g., we speculate that Bug 13 can be triggered along many execution paths and different call chains compared to the other bugs).

We confirmed that crash bucketing with T and $\widehat{T}$ result in *disjoint* buckets for multiple bugs in `SQLite` and `w3m`, and corresponds to the assumptions of Ideal Bucketing (i.e., zero overlap of crashing inputs of distinct bug fixes).

PHP. We applied SCB to one bug each in `PHP` v5 (CVE-2016-6292) and v7 (CVE-2016-10162). SCB improves over AFL's reports; the other configurations do not report duplicates.

R and **Conntrackd** both contain `strcpy` overflow bugs. The `R` bug is assigned CVE-2016-8714. For `Conntrackd`, we used our own 0-day `strcpy` bug that we found by prior fuzzing. Since no developer fix existed at the time, we manually debugged to develop a ground truth patch. A couple of months after disclosing the bug, we received a developer fix. The developer fix, our ground truth patch, and the approximate fix all bucket the same crashing inputs in our experiments.

As mentioned, Honggfuzz is particularly sensitive to changes in the stack, especially overflow vulnerabilities affecting the stack. Honggfuzz provides a way of blacklisting stack hashes to compensate,[9] but this option is disabled by default.

libmad. We also used a 0-day `memcpy` bug in `libmad` found by our own prior fuzzing. We similarly developed our own patch to perform the correct bounds checking on the length of bytes to copy. Interestingly, and prior to a fix, developers added a C `assert` statement before the `memcpy` call that checks the correct bounds. However, `assert` statements are not compiled in release versions and the bug still results in a `SEGFAULT`. We used the assert statement to inform an initial ground truth fix for checking the buffer bound. Once again, after receiving a developer fix, our own fix and the approximate fix bucket the same crashing input. Our deduplication gain is smaller for `libmad`, but remains precise. Our `libmad` example shows that approximate fixes extend to varieties of API calls in real-world bugs with little effort.

4.7 Discussion

Merits of SCB and approximate fixes. Our approach can be layered on top of existing fuzzer deduplication methods or as a drop-in replacement. In general, SCB opens the opportunity to parameterize bucketing using targeted program transformation. One advantage of automated patch generation is resilience to changes in unrelated code across revisions. Concretely, we can generate an approximate fix for any revision containing the bug. This is not true for static, developer written patches. As explained, we had to make careful effort to isolate and backport existing patches for a ground truth study.

Approximate fixes can also improve fuzzing performance and coverage [176]. Fuzzers are known to get stuck on shallow bugs that restrict execution past a memory corruption

[9] https://github.com/google/honggfuzz/pull/29

bug.[10] Our approach provides a lightweight, parameterizable solution to augment fuzzing behavior and overcome such obstacles.

Limitations. Our approach requires some up-front manual cost to parameterize automated behavior for generating approximate fixes. The complexity of the targeted bug class also bears on the difficulty of specifying appropriate semantic cues and patch templates, and various approximations will affect accuracy of semantic bug containment. We demonstrated, however, that conceptually simple patch templates and semantic cues work well for common bugs found by fuzzers in real-world programs. We speculate that the approach generalizes further to, e.g., division-by-zero, arithmetic overflows and use-after-free bugs. In general, we offer that one-off specification per bug class is competitive with per-fuzzer configuration that preclude fine-grained semantic control.

Our time spent selecting projects to evaluate was dominated by whether we could find ground truth fixes and crashing inputs. Though the sample is small, every project that satisfied these criteria has worked with our approach so far (i.e., we do not fail to find an approximate fix), modulo the need for incremental refinements in our approach (e.g., we added a preprocessing step that expands macros in `PHP` to discover null dereference syntax when the program crashes).

In our experiments we observe that both approximate and developer fixes address unique bugs with a single check. Conceptually, we can imagine a case where some buggy behavior (e.g,. a null pointer) may be checked once before branching on multiple paths, or alternatively along two different paths.

Depending on the transformation, crashing inputs may thus map to one or two buckets. Our approach for null dereferences currently follows the second strategy for bucketing (since we add the check close to where the dereference occurs). In our experiments, this matched the behavior of developer fixes. Note that we can extend our approach to use first strategy (i.e., by searching for branch points and inserting checks upstream) and even

[10] https://github.com/google/fuzzer-test-suite/tree/master/libxml2-v2.9.2

compare different strategies; exploring and comparing such transformation strategies holds interesting potential for future work.

Due to the difficulty of performing organic fuzzing campaigns for known bugs, we purposely generate a crash corpus by mutating existing crashing inputs. The generated crash corpus likely inflates the number crashes that an organic campaign would encounter. Nevertheless, the crash corpus serves as a useful upper bound to quantify precision of default deduplication techniques in fuzzers versus SCB.

We evaluate our approach on existing state-of-the-art fuzzers under default options and with small modifications to BFF and Honggfuzz. We recognize that deduplication can be tweaked and improved with additional parameters and post processing (e.g., stack hash blacklists), but we generally believe that fuzzers (like other tools), should run with sensible defaults.

4.8 Related Work

Our approach relates generally to existing work in identifying bug uniqueness and bucketing crashing inputs [66, 76, 78, 178]. Of particular interest, Chen et al. [66] propose a machine learning approach that ranks interesting test cases for compiler fuzzer output, and use fixing patches as ground truth to map crashing inputs to unique bugs. Semantic Crash Bucketing draws on the idea of using ground truth fixes to precisely identify unique bugs, obtaining similar precision to ground truth by automatically approximating fixes.

Recent work by Pham et al. [178] uses a clustering algorithm that relies on a semantic characterization of inputs as constraints over paths, with particular applicability to symbolic executors. Our approach also promotes a semantic characterization of bugs, but focuses on being sensitive to semantic properties of bugs themselves, rather than summarizing crashing inputs in terms of path constraints. Broadly, current techniques manipulate and analyze program input or otherwise instrument programs to obtain "read-only" behavior of

programs (such as input coverage [66], constraints on input [178], or crash callstack [157]) to group crashes. To the best of our knowledge, SCB is the first technique that appeals to *program modification* for precisely grouping crashing input in the absence of ground truth fixes.

Angelic debugging [63] seeks to modify programs by replacing expressions with values, which bears conceptual similarity to our approximating fixes for C library functions. Our problem focus differs, however: we seek accurate crash bucketing in the presence of duplicated or unreported bugs, while Angelic debugging seeks to fix failing test cases while preserving existing passing test cases.

In terms of program modification, our work relates to failure-oblivious computing [147, 184]. For instance, our rule-based application of fix templates shares similarities with the idea proposed by Long et al. [147], who modify a program so that a null dereference does not cause it to crash. The objective of failure-oblivious computing, however, is to make program execution resilient to crash-inducing effects of bugs such as null dereferences or divide-by-zero errors. By contrast, SCB seeks to *isolate* unique bugs by selectively applying program transformation, rather than providing an automatic catch-all technique for keeping a program running in the interest of resilience. Syntactic patches promote the benefit of "patches as better bug reports" [204] so that engineers can analyze semantic effects that influence crash bucketing. Peng et al. [176] show that applying program transformation while fuzzing can increase program coverage and reveal more bugs; while our approach focuses on accurate crash bucketing, our technique complements this recent idea.

Fault localization [69, 117, 140, 183] and automatic program repair [136, 141, 211] share similar high level goals for identifying bugs and automatically fixing them. This work is broadly complementary to ours, providing techniques that can assist with accurately identifying fault locations for patch placement, and appropriate program transformations for different bug classes.

4.9 Summary

We introduced Semantic Crash Bucketing, a way to perform crash bucketing using lightweight program transformation. We then developed an automatic approach that applies patch templates to approximate real developer fixes to perform crash bucketing. Our approach uses configurable rules (specified once per bug class) that instantiate and apply patch templates based on crashing behavior. Unlike coarse deduplication methods, rules and templates are sensitive to bug-specific semantic properties and crashing behavior. We developed approximate fixes for null dereferences and buffer overflows. We performed a ground truth study comparing SCB and approximate fixes to (a) true developer fixes and (b) crash deduplication of three state-of-the-art fuzzers (AFL, BFF, and Honggfuzz). Our results show that approximate fixes are competitive with crash bucketing precision of true developer fixes, and perform strictly better deduplication than all tested fuzzer configurations.

Our main insight drew on the principle that *fixing the bug* offers a precise way to associate crashing inputs with unique bugs. In general, finding true fixes is hard (Section 4.2), and we offer that approximate fixes can achieve desirable outcomes in behavior. Nevertheless, work in automated program repair seeks to *actually* fix bugs, and thus presents extensive value for techniques like Semantic Crash Bucketing, if successful.[11] To realize such possibilities, automated program repair techniques must mature to automatically reason deeply about *newly-discovered* bugs: their root causes, and the *semantic* changes that correctly fix the bug. Current APR approaches predominantly rely on tests to correctly fix bugs, and hence are not applicable to newly-discovered bugs found by automated techniques like fuzzing. In the next chapter, we consider a new approach that overcomes this limitation, and augment an existing static analysis to inform semantic-driven program repair.

[11] I.e., we can replace approximate fixes with even more accurate true fixes.

Chapter 5

End-to-End Static Automated Program Repair

This chapter presents a new automated program repair technique using separation logic. At a high level, our technique reasons over semantic effects of existing program fragments to fix faults related to general pointer safety properties: resource leaks, memory leaks, and null dereferences.

Automated program repair (APR) can be seen as a particular specialization of automated program transformation where a semantics-altering change must satisfy certain correctness conditions to be considered a true bug fix. Defining and performing both program changes and correctness validation are core challenges for achieving automated program repair. A successful approach, however, poses extensive impact. For one, APR is extremely valuable for directly improving software quality by fixing bugs [106]. When seen as another instance of targeted program transformation, APR *also* presents the ability to enable better automated reasoning (as exemplified by approaches in Chapters 3 and 4). We focus specifically on using semantic reasoning (informed by static analysis) to repair for previously unknown bugs, thereby overcoming limitations in test-driven approaches. Our approach is effective: we correctly fix undiscovered bugs in real-world code, six of which have been merged into

popular C and Java projects. We further show examples wherein bug fixes allow the static analysis to discover more true positive bugs.

5.1 Introduction

Software bugs are expensive and time consuming [56, 167], motivating research to find and fix them automatically. In Chapter 4, we presented additional motivation: the potential to improve dynamic fuzz testing results. Research in automated program repair holds promise for reducing software maintenance costs due to buggy code. Considered broadly, a *program repair* is simply a transformation that improves a program with respect to some abstract domain that describes correct versus incorrect program behavior. The vast majority of modern repair techniques (e.g., [120, 135, 146, 154, 164, 177]) use test cases to construct this domain. Tests are appealing because they are intuitive and widely-used in practice (more so than, e.g., formal specifications) and can straightforwardly indicate whether a given change improves the program in question (i.e., by turning a previously failing test case into a passing one).

However, tests are limiting in several ways, especially (though not exclusively) for APR. Writing good tests is nontrivial [167], rendering some real-world suites a weak proxy for patch correctness [181]. APR techniques and humans alike can overfit to even high quality tests, producing patches that do not generalize to the true desired functionality change [190]. Developers must write a deterministic, reproducible test case corresponding to a bug under repair to use test-driven APR. This use case is particularly applicable to, e.g., regressions, but is limiting for previously-unknown defects.

More fundamentally, tests are only suitable for finding and guiding the repair of certain kinds of bugs. Some bug types are simply difficult to test for in a finite, deterministic way [163]. Consider concurrency errors or resource or memory leaks: Figure 5.1a shows an

example memory leak from `Swoole`[1] (line 11), which may be fixed by adding a call to the project-specific resource allocation wrapper `sw_free` (Figure 5.1b). Alternatively, consider the code in Figure 5.2a, from `error-prone`.[2] The call to `resolveMethod` on line 3 can return NULL, leading to a possible null pointer exception on line 6. A developer committed a fix (with a test) that inserts a call to a custom error handler (`checkGuardedBy`, line 5). However, the *very same mistake* had been made on lines 10–13, in the same `switch` statement, but was not fixed for another 18 months. Even when bugs are deterministically reproducible, tests usually cannot identify recurring mistakes like this.

Finding and fixing these types of bugs motivate the use of quality assurance techniques beyond testing. Considerable recent progress has been made in expressive, high quality static analyses that can cost effectively find real bugs like these examples in real programs [13, 54]. Companies like Ebay [114], Microsoft [54], Facebook [61], and others publicize their development and use of such analysis tools in engineering practice. Some bug-finding tools, like `error-prone` or `Findbugs` [44] even provide "quick fix" suggestions for certain bug classes, simplifying the process of repairing them. Developers find such suggestions very useful [48], and indeed their absence has been identified as a barrier to uptake and utility of static analysis tools [116]. However, the suggestions present in current tools are simple, generic, and usually exclusively syntactic (e.g., recommending the addition of `static` or `final` modifiers to variable or function declarations). Moreover, they provide no semantic correctness guarantees.

We propose a new technique that automatically generates patches for bugs in large projects without a need for either tests or developer annotation, for a semantically rich class of bugs that can be identified by recent, sophisticated static bug-finding techniques based on separation logic [49, 50, 52, 82]. Our key insight lies in the novel way we adapt *local reasoning* [168, 170, 212], "the workhorse of a number of automated reasoning tools" [169], to

[1] `Swoole` is a popular concurrency library for PHP, https://github.com/swoole/swoole-src
[2] `error-prone` is an open-source static analysis tool, https://github.com/google/error-prone

```
1   swHashMap *hmap =
2     sw_malloc(sizeof(swHashMap));
3   if (!hmap) {
4     swWarn("malloc[1] failed.");
5     return NULL;
6   }
7   swHashMap_node *root =
8     sw_malloc(sizeof(swHashMap_node));
9   if (!root) {
10    swWarn("malloc[2] failed.");
11    return NULL; // returns, hmap not freed
12  }
```

(a) Memory leak: forgetting to free memory before return in `Swoole`.

```
1 #define sw_free(ptr)
2 if(ptr) {
3   free(ptr);
4   ptr=NULL;
5   swWarn("free");
6 }
```

(b) `sw_free` wraps a call to `free`.

Figure 5.1: Fixing a memory leak in the `Swoole` project

pull out relevant parts of program state, and then search for repairing code from elsewhere in the same program. At a high level, our technique searches for and adapts program fragments satisfying a generic, pre-specified semantic effect that address a given bug class (such as "if the file is open in the precondition of a program fragment, it should be closed in the postcondition" to address resource leaks, like the one shown in Figure 5.1a). These fix effects are generic, and need only be specified once per bug class. They are also language- and API-agnostic, which means our approach applies off-the-shelf to multiple source languages, and its patches automatically conform to the programming conventions in a given project (e.g., constructing a patch using a project-specific custom resource handler like `sw_free` for memory leaks, if available, or `free`, if not), without requiring any additional customization.

We instantiate our approach in a tool called FootPatch, an extension to the Infer [61][3]

[3] https://github.com/facebook/infer

```
case IDENTIFIER: {
  MethodSymbol mtd =
    resolver.resolveMethod(node, id.getName());
  // mtd may be null
+ checkGuardedBy(mtd != null, id.toString());
  return bindSelect(computeBase(context, mtd), mtd);
}
case MEMBER_SELECT: {
...
  MethodSymbol mtd =
    resolver.resolveMethod(node, id.getName());
 // same problem!
  return bindSelect(computeBase(context, mtd), mtd);
}
```

(a) Developers fixed the potential null pointer exception on line 6; 18 months later, they addressed the very similar bug on lines 10–12.

```
public static void checkGuardedBy(boolean cnd,
 String fmtStr, Object... fmtArgs) {
  if (!cnd) {
    throw new IllegalGuardedBy(String.format(fmtStr, fmtArgs));
  }
}
```

(b) `error-prone`'s custom guard handler.

Figure 5.2: Fixing a null dereference in Google's error-prone tool.

static analysis tool. Infer finds bugs by automatically inferring separation logic assertions over program statements. Infer reasons over a semantic, analysis-oriented Intermediate Language (IL), and applies to large, real-world programs written in multiple source languages. Separation logic can be used to encode a variety of desirable correctness properties [61, 81]. We situate our approach by extending analyses that find bugs related to the violation of pointer safety properties, the focus of Infer. In this dissertation, we restrict our focus to constructing additive patches for resource leaks, memory leaks, and null dereferences. We discuss directions for generalizing our technique in Section 5.5.

Our approach provides several important benefits over previous techniques for automatic patch generation or fix suggestion. By integrating directly into the static analysis workflow, our approach addresses different types of bugs than are handled by most dynamic APR

techniques, and can encourage the adoption of these robust static bug finding tools in practice [116]. Because of the way our approach uses compositional specifications, it can produce fixes that are significantly semantically richer than existing static "quick fix" suggestions. FootPatch can construct repairs that cross procedure boundaries, entail multiline fixes, and are robust to program-specific customization like wrapper APIs. These benefits are evident in the above examples, both of which FootPatch can repair automatically. Note, for example, a call to `checkGuardedBy` does not on its own constitute a repair, as it simply checks the results of a boolean expression. FootPatch can determine that the function implements the desired behavior because it searches over compositional function call results. Additionally, both bug fixes use custom resource wrappers, which are often desirable as fixes because they are consistent with the convention in other parts of the program. For example, `sw_free` wraps the `free` function, and performs additional, non-interfering operations by setting the pointer to null and logging a debug message. Finally, unlike previous repair techniques that build on more formal abstract domains [79, 126, 143, 175], our approach scales to real-world programs, automatically instantiates and applies its patches, and relies on a principled semantic treatment to argue patch correctness and prevent patch overfitting. That is: FootPatch demonstrates a promising and previously underdeveloped application of *end-to-end* identification and repair of previously undiscovered bugs in real programs.

Our contributions are the following:

- **Program Repair with Separation Logic.** We present a repair technique using separation logic to ensure a desired correctness property based on pointer safety. The abstract domain provides a basis for reasoning about explicit semantic effects introduced by patch fragments, and enables a formal argument for semantic patch correctness.
- **Repair Extraction and Application Formalism.** We formalize the search and

extraction of program fragments with respect to a repair specification, and define the conditions for patch generation and automatic patch application with respect to a detected bug.

- **Evaluation.** We present an evaluation on popular software projects. Our approach fixes 24 resource leaks, 7 memory leaks, and 24 null dereferences in popular Java and C programs, including 11 previously undiscovered bugs. We are unaware of any prior repair tool that supports multiple languages under a single analysis. Our implementation runs on big projects ($>$ 200 KLOC). Run time ranges from 7 seconds to 21 minutes per project, to perform both finding and fixing bugs within the project. For applicable projects, our experiments show that the majority of correct patches (53) are found by searching for repair candidates that are callgraph-local to the bug, and expanding repair search to the project globally fixes 2 additional bugs. We observe a false positive rate of only 7% for fixes. Moreover, we demonstrate anecdotal evidence that our technique can lead the original static analysis to discover more bugs after performing repair.

- **Open Source Repair Tool.** We implement our technique in a tool called FootPatch, built on top of the open source Infer static analyzer.[4]

Section 5.2 provides background theory underpinning our approach. Section 5.3 details our repair approach using separation logic. Section 5.4 evaluates FootPatch, a tool that implements our approach. Section 5.6 discusses related work; Section 5.7 concludes.

5.2 Preliminaries

We build our approach on top of the analysis engine used in Infer [59], an open source framework that uses separation logic and Hoare logic-style reasoning to find bugs at scale, particularly those related to heap or pointer errors. This analysis abstracts a program

[4] https://github.com/squaresLab/footpatch

$$E ::= x \mid nil \mid c$$
$$B ::= E = E \mid E \neq E$$
$$S ::= x := E \mid x := [E] \mid [E] := E \mid x := \textbf{new}() \mid \textbf{dispose}(E)$$
$$C ::= S \mid C; C \mid \text{if } (B)\ \{C\} \text{ else } \{C\} \mid \text{while}(B)\ [I]\ \{C\} \mid x := f(\overrightarrow{E})$$
$$\mathbb{P} ::= \cdot \mid f(\overrightarrow{E})\{\textbf{local}\ \overrightarrow{E};\ C; \text{return } E\}; \mathbb{P}$$

(a) A simplified Smallfoot grammar, for illustration.

$$H ::= E \mapsto E$$
$$\Sigma ::= H_1 \ * \ \cdots \ * \ H_n \mid \text{emp}$$
$$\Pi ::= B_1 \wedge \cdots \wedge B_n \mid \text{true} \mid \text{false}$$
$$P, Q ::= \Pi \wedge \Sigma \mid \text{if } B \text{ then } P \text{ else } P$$

(b) The assertion language grammar.

Figure 5.3

to an intermediate language, and then symbolically interprets it to find paths that may lead to particular property violations (like null pointer dereferences). This section outlines background concepts underpinning our approach and the analysis it extends: the abstract program model and separation logic assertions (Section 5.2.1), the frame inference procedure for discovering specifications (Section 5.2.2), and an overview of how the concepts fit together to find bugs statically in real-world programs (Section 5.2.3).

5.2.1 Program Model and Assertion Language

Infer and our analysis both begin by abstracting a source program in one of several languages (e.g., Java, C, C++) to the Smallfoot Intermediate Language (SIL) [49]. SIL is an intermediate analysis language that represents source programs in a simpler instruction set describing the program's effect on symbolic heaps. This is particularly suitable for static analyses that find bugs related to pointer safety properties.

Program model. Figure 5.3a shows a simplified SIL grammar to illustrate the overall

program model. A SIL program $\mathbb{P}$ is a set of procedures [50, 51, 60]. SIL procedures have single return values and do not access global variables. Procedures consist of a series of *commands* (C, in Figure 5.3a), which model actions that generate assertions over symbolic heaps (described next). The storage model is fairly standard [60, 173]: `Heap` is a partial function from locations `Loc` to values `Val` (for simplicity, locations are positive integers and values are integers): $\texttt{Heap} \stackrel{def}{=} \texttt{Loc} \rightharpoonup \texttt{Val}$. `Stack` is a function from variables to values $\texttt{Stack} \stackrel{def}{=} (\texttt{Var} \cup \texttt{LVar}) \rightharpoonup \texttt{Val}$. Variables are two disjoint sets: a set of program variables `Var` and a set of logical variables `LVar`. A program `State` is simply the combination of its heap and stack: $\texttt{State} \stackrel{def}{=} \texttt{Stack} \times \texttt{Heap}$.

Assertions. SIL commands primarily capture effects over symbolic heaps, which comprise the abstract domain for detecting faulting conditions (e.g., memory leaks, resource leaks, null dereferences).[5] These effects can be described via pre- and postconditions expressed in separation logic, which decorate the SIL commands accordingly. Figure 5.3b shows the assertion language grammar. The grammar encodes heap facts using points-to heap predicates over program and logical variables (i.e., $E \mapsto E$). Heap predicates are considered "separate" sub-heaps (or heaplets), whose separation is denoted by the *separating conjunction* $*$ (read "and separately"). The separating conjunction implies that the two sub-heaps are disjoint. Pure boolean predicates of the form $B_1 \wedge \cdots \wedge B_n$ assert conditional facts over heap predicates (e.g., $E = nil$).

Given our focus on repair, assertions are relevant insofar as they describe semantic effects of statements (as predicates) on the heap. We denote by $E \Mapsto \mathit{Alloced}$ a predicate $\mathit{Alloced}$ on E for allocated memory (e.g., we may represent it in the grammar by the assertion $E \mapsto x \wedge \mathit{true}$ for x fresh). For simplicity, this notation may assume a program variable evaluates to a heap location (such as `hmap`); we do this with the understanding that stores and heaps are typically treated separately in the storage model [60]. As a practical

[5] Symbolic heaps enable a decidable proof system for entailments under a prescribed semantics [49]; we elide details for brevity.

matter, Infer defines additional predicates *File* and *Memory* to distinguish file resources from memory. These predicates are attached via function models (e.g., `fopen` adds the *File* predicate; `malloc` adds the *Memory* predicate). We use our convenience notation $pvar \Mapsto \langle \mathit{Alloced, File} \rangle$ to express allocated memory with the attached *File* predicate. We express freed memory with a corresponding assertion $pvar \Mapsto \langle \mathit{Freed, File} \rangle$. With respect to Figure 5.3b, a freed operation corresponds to assigning a variable "emp". As a final practical matter, Infer provides an *Exn* heap predicate to indicate exceptional behavior for some exception e. We express this as $_ \Mapsto \langle \mathit{Exn}\ e \rangle$.

5.2.2 Frame Inference

A key novelty in our work is the way we extend *frame inference* [50] to find bug repairs; we thus briefly overview frames and frame inference in this context. Infer's separation logic-based analysis uses Hoare-style reasoning to find specification violations. It does this at scale by summarizing the effects of individual terms in a procedure, and then composing them into procedure-level specifications. *Local reasoning* [168, 170] is used to summarize the effects of those individual terms. Local reasoning is enabled by the fact that a program command can often affect only a sub-part of the heap. For example, the statement `hmap = sw_malloc(sizeof(swHashMap))` is modeled as affecting only `hmap`; the rest of the heap is unaffected by the allocation. The fact that sub-states can change in isolation is modeled by the separating conjunction. The unchanged part of the heap for a command is its *frame*; the parts of the heap that a command reads from, or writes to, is known as its *footprint*.

Thus, *frame inference*, which automatically infers command frames such that they can be composed efficiently into procedure summaries [60][6] is key to a number of automated

[6]For completeness, this compositional analysis also infers anti-frames, or the missing parts of the heaps state. Anti-frame inference allows the analysis to deal with unknown calling contexts and increases precision by propagating intermediate results. Anti-frame inference is not critical to our approach.

reasoning tools [169]. By discovering the unchanged heap portion of an operation, frame inference discovers footprints, expressed as small specifications of program terms [50, 81, 165].

More formally, the Frame Rule codifies the notion of reasoning over local behavior:

$$\frac{\{P\}\ C\ \{Q\}}{\{P\ *\ F\}\ C\ \{Q\ *\ F\}}\ \textsc{Frame Rule}$$

The Frame Rule allows analysis of a command C with a specification $\{P\}\ C\ \{Q\}$ and a heap state H to proceed, without considering unaffected parts of H (the frame F), if it can be separated into parts $\{P\ *\ F\}$. Frame inference [50, 60, 80] discovers a frame F that allows the Frame Rule to fire, enabling local reasoning over the footprint. We can summarize frame inference as follows. Let $\textsc{Frame}(H, S)$ be the frame inference procedure that returns a frame F (if it succeeds) for a given heap state H and a specification S of a command (expressed in the grammar of Figure 5.3b). The procedure consists of a proof system using subtraction and normalization rules to partition the heap H into $S\ *\ F$. We refer to previous work for the complete algorithm and proof system [50].

5.2.3 Finding Bugs Using Separation Logic

The previously described reasoning enables scalable, compositional static bug finding with minimal developer effort (in the form of, e.g., annotations or customization) over real-world code bases. Infer implements these by ideas by converting source programs into SIL, and then infers specifications (described as symbolic heaps) for SIL program fragments. It discovers bugs by symbolically executing SIL commands over symbolic heaps, according to a set of operational rules that update symbolic heap assertions [50]. The general goal of this static analysis is to discover program paths with symbolic heaps that violate heap-based properties.

Infer currently supports detecting a variety of heap-related bugs using separation logic:

resource leaks, memory leaks, null dereferences, as well as experimental support for buffer overflows, thread safety, and taint-style information flow bugs (e.g., detecting SQL injections for unsanitized values) [11, 12]. While we believe our approach generalizes to these other bug types, we focus here on pointer-safety properties of heaps for repair, specifically resource leaks, memory leaks, and null dereferences, as the support for them in Infer is most mature.

To illustrate, consider the memory leak described in Figure 5.1a. Infer discovers this error by identifying the path through line 11 where the variable `hmap` is allocated but not freed before becoming dead. When it discovers such an error, the symbolic interpreter enters a special state, `fault`. We denote this formally by $C_\ell, \sigma \rightsquigarrow \texttt{fault}$, meaning that the interpretation step $\rightsquigarrow$ for instruction C at location ℓ in symbolic state σ results in a `fault`. For this example, `hmap` is still allocated in the symbolic heap (denoted by the predicate *Alloced*, i.e., $\{\texttt{hmap} \Mapsto \mathit{Alloced}\}$) at the location $\ell = 11$ when it becomes dead: $\texttt{return}_\ell, \{\texttt{hmap} \Mapsto \mathit{Alloced}\} \rightsquigarrow \texttt{fault}$.[7] At this point, our approach takes over from the bug-finding analysis, to seek a potential fix.

5.3 Repair with Separation Logic

This section presents our program repair technique using separation logic. Section 5.3.1 formalizes our notion of repair with respect to heap-based property violations. In Section 5.3.2 we formalize the search procedure to discover candidate patch code, drawn from existing program fragments (i.e., from elsewhere in the program under repair). In Section 5.3.3 we describe the application of candidate fragments in source code, in terms of where to introduce a change and how we filter out invalid candidates. We illustrate throughout by referring to the motivating example in Figure 5.1.

[7] For illustration, we consider only the predicate on `hmap`, ignoring the rest of the state.

5.3.1 Formulating Repair

Fundamentally, any program transformation (for repair or otherwise) is composed of either one or a combination of two primitive operations: addition and removal of program terms. Taking separation logic as the abstract domain, a bug fix corresponds to a program transformation that leads to a *fault-avoiding interpretation* in the analysis with respect to the property in question. We presently consider only *additive* program transformations, and do not perform removal operations, because the types of bugs we consider are typically caused by the *lack* of certain operations on explicit heap content (e.g., resource release, freeing memory, or checking nullness of return values). For instance, developers often forget to insert missing checks [14, 206]. Bug-fixing changes for these types of bugs thus correspond to inserting missing statements (e.g., checks, initializations, or cleanup handlers). Other types of repairs are certainly valuable, in this domain and others, but we leave discovery of them to future work.

The central repair question is thus: Does there exist a fragment $?C$ that can be used to transform the program such that the `fault` state is no longer triggered? We express this formally as follows:

Definition (Program Repair)**.** *Let* $\mathbb{H}_{Bad}$ *be the heap configuration that results in a fault under interpretation of a command* C *at location* ℓ. *A repair is an additive transformation* T *on a program* $\mathbb{P}$ *that transitions the fault-inducing predicate in the heap state* $\mathbb{H}_{Bad}$ *to a heap state* $\mathbb{H}_{Good}$ *that preserves a fault-avoiding interpretation for* C *at* ℓ. *A repair satisfies:*

$$C_\ell, \mathbb{H}_{Bad} \rightsquigarrow \texttt{fault} \Longrightarrow T_{\ell'}, \mathbb{H} \overset{*}{\rightsquigarrow} C_\ell, \mathbb{H}_{Good} \not\rightsquigarrow \texttt{fault}$$

The additive transformation T is a program fragment (operating on the program heap $\mathbb{H}$ at some point ℓ') that induces a desired *fix effect* on the heap, producing $\mathbb{H}_{Good}$ at ℓ. The fix effect precisely defines what it means to avoid the `fault` state.

We encode the transformation T (i.e., the fix effect) as a Hoare-style triple that we call a

Repair Specification. We specify T as two singleton heaps (i.e., a single points-to relation as in Figure 5.3b): F mapping to a *fixable* predicate over a placeholder variable $pvar$ in the precondition, and a corresponding F' mapping to a *fixed* predicate over that placeholder $pvar$ variable in the postcondition. F and F' express the desired symbolic transition on the abstract predicates. For example, $F = \{pvar \Mapsto Alloced\}, F' = \{pvar \Mapsto Freed\}$, specifies a generic fix effect for memory leaks over placeholder variable $pvar$. Note that such fix effects are generic to entire bug classes. By expressing repair over the abstract domain describing what the code *does*, this fix effect specification approach is multi-language and resilient to syntactic customizations (like APIs or wrapper functions).

Definition (Repair Specification)**.** *A* Repair Specification *R is a specification containing a program term* ***repair fragment*** *C_R (a command C in the Smallfoot grammar) that effects a state transition from an error heap configuration F to a fixed heap configuration F', denoted $\{F\}\ C_R\ \{F'\}$, via an atomic update of an abstract predicate.*

In its current form, we manually specify appropriate F and F' corresponding to the general bug classes in question. Defining a mechanism to automatically determine F and F' (e.g., by formally deriving it from a violation reported by an analysis) is an interesting and plausible research direction that we leave for future work. Note however, that fix effects are generic to an entire bug class, and thus need only be specified once per analysis type to be applied to a given program. Moreover, the static analysis provides a degree of confidence in the choice of F and F': a poor choice will not ensure a fault-avoiding interpretation for a particular fault, and will be detected by analysis of the transformed program.

Our implementation provides default fix effects for the bug classes we consider that suffice for many off-the-shelf projects, requiring customization only when a project uses a particular or unique paradigm for handling, e.g., custom exceptions. Extending our technique to new static checkers (employing automated semantic reasoning as found in Infer) could similarly involve specifying default fix effects for patching them, eliminating

the need for developer-provided specifications for many real-world projects.

5.3.2 Searching for Repairs

Repair Queries. Different automated techniques can discover repairs, including program synthesis [126] or syntactic program mutation [135]. Our technique does so by searching over existing program fragments, which can often exhibit the desired semantic effects to fix faults [126, 134, 146]. Using existing program fragments also preserves program-specific syntactic structure and semantic shape that accompany a fix, and may decrease the risk of overfitting repairs [120]. We thus search for program fragments C_R in the Smallfoot IL from across the rest of the program with a Repair Query. A Repair Query encodes the desired semantic transition and returns satisfying Repair Specifications.

We illustrate this overall framing with respect to our running example in Figure 5.4: The computation in (1) shows the semantic change that must be induced by some $?C_\ell$ to preserve a fault-avoiding interpretation, fixing the memory leak bug in question. The computation in (1) informs the repair question labeled (2) in Figure 5.4. The specification in (2) describes a desired program fragment $?C$ that induces the desired fix effect on symbolic variable *pvar*, which is allocated on precondition to $?C$, and freed on postcondition. Note that this fix effect is flexible, and could describe fix code such as `free` modeled in generic C or the custom `sw_free` function. A Repair Query seeking to repair a file resource leak, on the other hand, could find fragments such as `close` or `fclose` in C, or `f.close()` for a file `f` modeled in Java. Although the fix effects must be either inferred or specified, their portability across multiple programs and languages amortizes the manual burden and represents an important improvement over the labor required to use test-based APR techniques, which require a triggering test case per bug under repair.

We call (2) the Repair Query, a Hoare triple containing a "hole" for an instruction C that induces the desired semantic change under the standard partial correctness interpreta-

$$?C_{\ell'}, \{\texttt{map} \mapsto \mathit{Alloced}\} \rightsquigarrow \texttt{return}_{\ell}, \{\texttt{map} \mapsto \mathit{Freed}\} \not\rightsquigarrow \texttt{fault} \quad (1)$$

$$\{\mathit{pvar} \mapsto \mathit{Alloced}\} \;\; ?C \;\; \{\mathit{pvar} \mapsto \mathit{Freed}\} \quad (2)$$

Figure 5.4: Modeling repair search.

tion. For example, the fixing fragment `sw_free(map);` corresponds in the IL to a command C of the form $\{\texttt{map} \mapsto Alloced\}\ call(\cdot)\ \{\texttt{map} \mapsto Freed\}$, with the concrete program variable `map` bound to $pvar$.

Semantic search constraints. A Repair Query expresses a *symbolic* transition on the abstract predicates, providing a basic structure that expresses fixes in terms of desired semantic heap properties. So far, our example Repair Query specifies C_R program fragments that perform strictly the desired symbolic transition, disallowing any C_R that may introduce extra semantic effects. However, as a practical consideration, it may be desirable to search for program fragments that introduce extra semantic effects in addition to the fixing effect. For example, `sw_free` performs cleanup and logging in addition to resource freeing; in other cases, such side effects may be benign, or undesirable.

Fortunately, separation logic allows us to elegantly relax strict Repair Queries by explicitly allowing for and then capturing extra semantic effects beyond the desired fix effect. This produces a repair search method that can be parameterized by extra semantic effects, explicitly partitioning such effects from the fixing effect (absent a priori knowledge to the fixing effect). Such a search returns relaxed Repair Specifications of the form $\{F\ *\ P\}\ C_R\ \{F'\ *\ Q\}$, where (potential) extra semantic effects are bound in the pre- (P) or postcondition (Q) respectively. Finding satisfying Repair Specifications comes with an analogous extension of a Repair Query, i.e., $\{F\ *\ P\}\ ?C\ \{F'\ *\ Q\}$.

Pertinently, we can use *frame inference* in a novel way to solve two important (and distinct) purposes in the program repair search problem: (1) Check whether a given specification satisfies a repair query, and (2) discover the extra semantic effects not part

of the fix. That is, suppose some command C has a footprint, expressed as $\{S_P\}\ C\ \{S_Q\}$. Our first goal is to check whether the footprint satisfies the REPAIR QUERY, which we do with respect to corresponding pre- and postconditions F and F'. Our second goal is to partition the footprint into the fixing transitions and extra semantic effects for pre- and postconditions, respectively.

Our key insight is to perform the frame inference procedure FRAME (Section 5.2) on the *footprint* precondition (resp. postcondition) with respect to the REPAIR QUERY precondition (resp. postcondition). Formally, C is a candidate repair fragment when the following entailments hold:

In the precondition:

$$\textsc{Frame}(S_P, F) = P \Longrightarrow S_P \vdash F\ *\ P \tag{5.3}$$

Respectively, in the postcondition:

$$\textsc{Frame}(S_Q, F') = Q \Longrightarrow S_Q \vdash F'\ *\ Q \tag{5.4}$$

We achieve (1) because the entailment does not hold (frame inference fails) when F is not satisfied by the query. If frame inference succeeds, it pulls out P (resp. Q), discovering the extra semantic effects in the footprint of C, achieving (2). Our approach is sound and decidable by the frame inference procedure [50].

Assume $\mathfrak{F}$ contains the specifications inferred over all individual commands in a program. Algorithm 1: FINDCANDIDATES(R,$\mathfrak{F}$) returns all candidate REPAIR SPECIFICATIONS for query R. In Line 4 and 5 of Algorithm 1, we use the frame inference procedure FRAME, to match candidate fragments with a repair query as the conjunction of matching pre- and postconditions. Note that the procedure returns the entire REPAIR SPECIFICATION because we use assertions in the precondition for repair application (Section 5.3.3).

Algorithm 1: Find Candidate Repair Fragments

```
1  Match(S,R);
2      let {R_F} ?C_R {R_F'} = R in
3      let {S_P} C {S_Q} = S in
4      if Frame(S_P,R_F) = R_P ≠ fail ⇒ S_P ⊢ R_F ∗ R_P
5      ∧ Frame(S_Q,R_F') = R_Q ≠ fail ⇒ S_Q ⊢ R_F' ∗ R_Q then
6      |   return {R_F ∗ R_P} ?C_R {R_F' ∗ R_Q}[C/?C_R]
7      else
8      |   return fail
9      end
10
11 FindCandidates(R,𝔉);
12     ℭ := ∅ // Candidates
13     foreach S ∈ 𝔉 do
14     |   if Match(S,R) = 𝒞 ≠ fail then
15     |   |   ℭ := ℭ ∪ 𝒞
16     |   end
17     end
18     return ℭ
```

5.3.3 Applying Repairs: from Logic to Programs

A Repair Specification in the abstract domain must be translated to a syntactic program fragment in the source program. Every intermediate language (IL) instruction corresponds to a line in the original program source. When a candidate program fragment is translated from IL to source, we rename the program variable bound to *pvar* in the fixing fragment to that of the fault-inducing variable, if necessary. This substitution is the only syntactic modification that we make to discovered program fragments. Beyond this renaming, we must

1. decide where to insert repairs and
2. check translated code for compatibility given associated heap assumptions, available variables, and type information (available in the IL).

Determining repair location. Repair techniques typically rely on dynamic fault localization techniques to determine placement or manipulation of code [39]. By contrast, we rely on the static analysis to localize faults for repair: When the symbolic interpreter enters

a `fault` state, it provides a location ℓ where the fault occurs.

In general, the fault class bears on the choice of where to apply a repair fragment. This observation is consistent with motivating different choices of where to apply patches automatically for approximate fixes in Chapter 4.[8] To avoid null dereferences, our intuition is that a developer might typically expect a change before the null dereference (e.g., a check). This is consistent with our results in Chapter 4, where ground truth developer fixes for null dereferences predominantly consist of a check preceding the bug-inducing dereference. For resource leaks, we alternatively expect a change after the point at which the resource is last used (e.g., closing a file). Using Infer, a null dereference reports the location ℓ immediately preceding the dereference. For resource leaks and memory leaks, the location ℓ is the point where the resource (resp. memory) goes out of scope without being closed (resp. freed). We make the design choice to insert a repair fragment C_R directly preceding the program term at location ℓ, satisfying

$$C_{R_{\ell'}}, \mathbb{H}_{Bad} \rightsquigarrow C_{\ell}, \mathbb{H}_{Good} \not\rightsquigarrow \texttt{fault}$$

We refer to the example in Section 5.1 to motivate our choice. Our approach places the `sw_free` call at line 11 in Figure 5.1a. The first motivation is that repair location is typically restricted to few alternatives; the only other correct choice of placement is line 10. The location ℓ thus offers the most immediate (and sometimes strictly) correct choice. The second motivation is that among multiple placement locations, the semantic effects of repair remain the same. For example, placing `sw_free` at line 10 in Figure 5.1a. produces the same effect since the `swWarn(...)` statement does not affect the heap. Thus, although there is no universally correct choice for repair placement, our domain of repair benefits from a general strategy that preserves semantic correctness. Determining the ideal placement of patches with respect to non-semantic attributes may vary by context, and may be subject

[8] cf. the difference between null dereferences (Section 4.4.2) and buffer overflows (Section 4.4.3).

to stylistic conventions and human judgment that go beyond the scope of this work.

Determining patch compatibility. Patch compatibility determines whether we *can* insert a syntactic program fragment at a particular program point (ℓ, in our case). FootPatch performs two compatibility checks. The first addresses bugs that can have multiple candidates. For example, memory may be freed by the standard C library `free` call, or a wrapper function such as `sw_free` as in Listing 5.1b. When multiple candidate fragments in SIL matches the desired semantic effect (based on predicates that do not consider types), FootPatch prioritizes patch generation by matching type information of a candidate fragment's variable and the fault-inducing variable. This means that FootPatch prefers `sw_free` over `free` in our example, because we can infer the type of `hmap` to be `swHashMap *`, which matches the same type of `map` in the candidate fixing fragment `sw_free(map)`. FootPatch falls through to matching candidates with generic types (i.e., `void *` for C) if it cannot match specific types. Although typing information could be used to refine patches for null dereferences, we ignore typing information in this case, since any object can be compared to null.

Recompilation serves as our second compatibility check, ensuring (a) that syntactically malformed patches fail (due to poor IL-to-source translation or fault locations), and (b) that program fragments with unbound free variables are invalidated. For instance, the fix for the bugs in Figure 5.2 binds to the variable `id` that is in scope. If `id` were *not* in scope, patch generation fails. Relatedly, FootPatch allows capture of program variables (beyond the fault-inducing variable) in fixing fragments if they are available (such as `id`, which is in scope).

Syntactic barriers. Surprisingly, the need for syntax manipulation can present a final hurdle to automated application in spite of the confidence inspired by a semantic-driven repair approach. That is, syntax concerns persist even after extracting plausible program fragments, reasoning about their semantic effects, determining repair location, and checking

patch compatibility. For example, consider the following code, where a leaked file pointer is reported for the variable `fp`:

```
if (fgets(line, 128, fp) == NULL)
  return; // Infer reports that fp is leaked at this program point.
```

Suppose we have identified `fclose(fp);` as a possible program fragment that satisfies a repair. Inserting the `fclose` statement poses a problem: we cannot naïvely insert it in the correct position, right before the `return;` statement. The problem is that braces are optional for `if` statements when the if-body contains only one statement. In the example above, the braces are omitted. When more than one statement belongs to an if-body, braces become mandatory. When we add the `fclose` statement, it needs to be in scope of the if-body, and we must also preserve the return statement. The naïve insertion will cause the program to always return unconditionally. Thus, we must add braces:

```
// Wrong: return always executes.
 if (fgets(line, 128, fp) == NULL)
+  fclose(fp);
   return;
```

Naïvely adding the fixing statement is wrong because optional braces are omitted.

```
// Correct: braces preserve scoping.
 if (fgets(line, 128, fp) == NULL) {
+  fclose(fp);
   return;
 }
```

The desired fix requires a syntactic change that adds braces.

Again, regular expressions are not generally expressive enough to easily and robustly perform this change (cf. Section 2.2). Ideally we want to match the end of the if condition (which can contain nested parenthesized expressions) and then surround the body in braces. A reliable solution would be to use `Clang` for refactoring, which can correctly add the braces around the if-body. The problem is again that Clang is susceptible to failure on certain C files (cf. Section 4.4.2), and is also too heavyweight a solution (it requires an AST or refactoring plugin). `Coccinelle` [132] offers a middle ground, but is C-specific; changing the syntactic pattern above is also needed for similar fixes in Java, which `Coccinelle` cannot

handle.[9] Naturally, we use Comby, which was developed with these interests in mind: small, targeted syntactic changes that work easily for multiple languages. The rewrite pattern we use is simple, and adds braces to the if-statement (Figure 5.6). We then insert our fixing fragment.

Match template

```
if (:[1])
  return;
```

Rewrite template

```
if (:[1]) {
  return;
}
```

Figure 5.6: The rewrite pattern adds braces around all if-statements that only have a return in the body. Note that whitespace is matched insensitively (i.e., indentation level or spacing does not matter; see Section 2.4.1).

There is flexibility in applying the above transformation. For one, we can preprocess the entire project before running the repair. This can create many changes that are unrelated to the fix, however, and may not be desired by project maintainers. Alternatively, we can preprocess the entire project, perform repair, and then reverse the transformation (i.e., remove braces) for all if statements that contain only a return. The solution we opt for approximates the textual range of the if-fragment for candidate repairs (when applicable), then attempts the brace-addition transformation, followed by insertion. As a practical matter, brace addition can shift lines of code with respect to analyzer-reported lines; thus, transformations are applied starting toward the end of the file, working upward.

5.4 Evaluation

This section describes the results of using FootPatch to fix bugs in real-world programs. Section 5.4.1 describes our experimental setup. Section 5.4.2 describes overall repair results, where we use FootPatch to fix 55 bugs in 11 projects. In Section 5.4.3 we discuss patch quality, and particularly its relationship to Infer's underlying static analysis and our other

[9] Moreover, similar syntactic features also exist in functional languages, like OCaml

design decisions. Section 5.4.4 analyzes FootPatch's repair discovery for resource leaks, memory leaks, and null dereferences in the context of our general formalism (Section 5.3). Section 5.5 provides limitations and further discussion.

5.4.1 Setup

Implementation. Based on the techniques described above, we implemented an automatic bug repair tool called FootPatch extending the Infer static analyzer. FootPatch works on multiple source languages (we evaluate on programs in C and Java; Infer also supports C++ and ObjectiveC) because, like Infer, it reasons over programs translated into SIL. FootPatch uses the same predicates in the IL irrespective of source language, meaning it can apply directly to new languages as support for them is added to Infer.

$$\{\ \mathit{pvar} \mapsto \mathit{Null}\ \}\ ?C\ \{\ _\ \mapsto \mathit{Exn}\ e\ \} \tag{5.5}$$
$$\{\ \mathit{pvar} \mapsto \langle\ \mathit{Alloced},\ \mathit{File}\ \rangle\ \}\ ?C\ \{\ \mathit{pvar}\ \mapsto \langle\ \mathit{Freed},\ \mathit{File}\ \rangle\ \} \tag{5.6}$$
$$\{\ \mathit{pvar} \mapsto \langle\ \mathit{Alloced},\ \mathit{Memory}\ \rangle\ \}\ ?C\ \{\ \mathit{pvar}\ \mapsto \langle\ \mathit{Freed},\ \mathit{Memory}\ \rangle\ \} \tag{5.7}$$

Figure 5.7: Repair Specifications.

FootPatch uses three general specifications for repairing null dereferences, resource leaks, and memory leaks; Figure 5.7 presents the specifications in the notation explained in Section 5.2.1. The specifications are implemented as OCaml functions that match pre/post conditions on heap state and predicates (including Infer's datatypesfor *File*, *Memory*, and *Exn* predicates). Note that this specification mechanism is not intrinsic to the technique (i.e., it is possible write a DSL for expressing such specifications). FootPatch performs type matching for determining patch compatibility, (Section 5.3.3) by extending specifications (6) and (7) in Figure 5.7 with an optional guard clause "`when` $\mathit{pvar.type} = t$" if we can determine the type t of a faulting variable. In general, since simple expressions do not model semantic effects of interest to the bug types in question, we restrict repair queries to calls and branch statements in the IL.

Data and experimental parameters. We ran our experiments on an Ubuntu 16.04 LTS server, with 20 Xeon E5-2699 CPUs and 20GB of RAM. Table 5.1 includes projects which (a) successfully built on our system, (b) could be analyzed by Infer, and (c) generated patches. Our project selection represents a convenience sample, intended to substantiate our claims about FootPatch applicability to real bugs in real and actively developed open-source systems; we include discussion of sources of failures and other technique limitations in Section 4.5. We evaluate on 8 C programs and 3 Java programs averaging 64 KLOC. We initially developed FootPatch based on existing bugs in `error-prone` and `jfreechart` and new bugs found in `swoole`. The rest of the projects are C and Java repositories on GitHub that are either (a) randomly sampled from the top one thousand most popular repositories (by user favorites, or stars, in November, 2016) for C and Java respectively, or (b) contain any combination of the terms "leak", "resource", "memory", "file", or "fix" in their commit messages. Projects in our sample are excluded if they fail to compile in our environment, if they cannot be analyzed by Infer, or if FootPatch did not find patches (either because no bugs were found or due to some other failure).

We evaluated FootPatch in two modes to characterize its search behavior. In *callgraph-local* mode, FootPatch searches over candidates from the callgraph of functions where Infer reports a bug. This mode tracks whether candidates are found local to the function containing the bug, local to the file containing the bug, or from an external file. In *global mode*, FootPatch searches over all (disjoint) callgraphs. Global mode subsumes callgraph-local mode and, in our experiments, only discovers additional repair candidates in external files not included in the local callgraph. Global mode is naturally more time-intensive, but may identify additional patches; comparing the two modes allows a more precise understanding of the importance of locality in searching for bug fixes within a given program. In all results discussion, ΔGL indicates the increment searching globally has over searching locally.

Project	Lang	KLOC	Time (s)	ΔGL	Bug Type	Bugs	Max Cands	ΔGL	Fixes TP	Fixes ΔGL	FP	ΔGL
Swoole	C	44.5	20	+83	Res. Leak†	7	1	+6	1	+2	0	+0
					Mem. Leak†	20	3	+0	6	+0	3	+0
lxc	C	63.0	51	-	Res. Leak	3	5	-	1	-	0	-
					Mem. Leak	8	13	-	0	-	1	-
Apktool	Java	15.0	584	+92	Res. Leak†	19	3	+2	1	+0	0	+0
dablooms	C	1.2	9	+0	Res. Leak†	7	2	+0	7	+0	0	+0
php-cp	C	9.0	20	+5	Res. Leak†	4	3	+1	1	+0	0	+0
armake	C	16.0	10	+13	Res. Leak†	5	7	+4	4	+0	0	+0
sysstat	C	24.9	28	+10	Res. Leak	1	10	+0	1	+0	0	+0
redis	C	115.0	79	+121	Res. Leak†	8	8	+10	6	+0	0	+0
rappel	C	2.1	7	+3	Mem. Leak†	1	6	+0	1	+0	0	+0
error-prone	Java	149.0	262	+602	Null Deref	11	66	+0	2	+0	0	+0
jfreechart	Java	282.7	1,268	-	Null Deref	53	221	-	22	-	0	-

Table 5.1: Bugs repaired with FootPatch. "**Bugs**" is the number of bugs detected by Infer's static analysis. "**Max Cands**" is the maximum number of IL repair candidates for the bug (pre-compatibility check). "**Fixes**" are the number of unique patches fixing true positive bugs (post check). "**FP**" is the number of false positive bugs reported by Infer (determined by our human inspection) for which FootPatch produces a patch. † indicates one or more fixes for previously undiscovered bugs. "**ΔGL**" is the change in associated column when using the global search space.

5.4.2 Repair Results

Table 5.1 show results for each project. "Bugs" indicates the number of bugs detected by Infer of the given type. It is possible for multiple semantic fragments to repair each type of bug, found at different locations in the SIL callgraph. For each bug type per program, "Max Cands" shows maximum number of IL repair candidates before the compatibility check which determines whether a patch candidate is suitable syntactically. FootPatch emits the first compatible patch produced from the candidates. "Fixes" shows the number of unique patches that fix true positive bug reports. Conversely, Infer may report false positives. The "FP" column shows the number of unique patches that we generate for false positive bug reports. It is out of scope for FootPatch to distinguish true and false positive bugs; we manually inspected bug reports and patches to qualify the overall utility for building automatic repair on top of an industrial static analyzer (i.e., false positives are few, and most patches correctly fix true bugs). We elaborate on patch quality and correctness in Section 5.4.3.

In total, we discover 24 fixes for resource leaks, 7 fixes for memory leaks, and 24 fixes for null dereferences. The "Time" columns of Table 5.1 shows *total* time required to both find and patch all bugs of all types considered in that program. Performance ranges from 7 seconds to 22 minutes over all projects (note that FootPatch performance is intertwined with Infer's analysis time). The `jfreechart` experiment failed to terminate in global mode because the Infer analysis phase ran out of memory; `lxc` failed to build in the global configuration.

Global mode (ΔGL) discovers only 2 additional fixing patches. These patches fix resource leaks due to discovering a `close`-like function that is not present in the local search. Our results suggest that localizing search for fixing fragments is an effective strategy for repair. This is consistent with empirical results that suggest that code is redundant locally, especially within a module [193].

```
1      fp = fopen(rdbfilename,"r");
2      ...
3      if (memcmp(buf,"REDIS",5) != 0) {
4          rdbCheckError("Wrong signature trying to load DB from file");
5  +       fclose(fp);
6          return 1;
7      }
8      rdbver = atoi(buf+5);
9      if (rdbver < 1 || rdbver > RDB_VERSION) {
10         rdbCheckError("Can't handle RDB format version %d",rdbver);
11 +       fclose(fp);
12         return 1;
13     }
14    ...
```

Figure 5.8: `fp` can be leaked on at least two paths (lines 6 and 12), but Infer short circuits the analysis and only reports the leak on line 6 by default. With FootPatch, the leak is fixed at line 6, allows Infer to find the another resource leak, which is then also fixed at line 11.

5.4.3 Patch Quality

Patch correctness and success. All produced patches ensure a fault-avoiding interpretation; in practice, we apply each patch generated by FootPatch and rerun the static analyzer to see if the patch removes the bug (all did). Where possible, we ran a project's test suite after applying our patches to validate that our changes do not break tests. We successfully ran the test suites for `Apk-tool`, `armake`, and `error-prone`, which pass. Two projects contained no tests, and the remaining six test suites could not be successfully configured/built.

A "fix" in Table 5.1 produces a patch that addresses a *true positive* bug report from the static analysis. To be useful in practice, analyses approximate [142]. Infer is no exception, and it sometimes skips inferring specifications for a function due to an analysis timeout, continuing with partial results. This can lead to false positives. FootPatch uses Infer's results to perform patching, and cannot distinguish between true and false positives (if it could, it would be a better analysis than Infer. It is especially interesting, however, that our approach for tailoring programs (Chapter 3) presents the potential of removing Infer's false positives when bug reports are used for performing repair. A manual inspection of

Infer's reports indicate that its false positives generally arose when it failed to analyze loops or clean up functions due to time out. Due to the low number of false positives, we did not investigate deeply how tailoring may address these in our experiments. Due to the complexity of `jfreechart` it is difficult to precisely determine how many of the Infer-reported null dereferences are false positives. However, our manual inspection did not reveal obvious errors in reasoning behind the produced patches. False positives that produced patched bugs are listed in column "FP" of Table 5.1. In general, the false positive rate is low, on the order of 7%, where fixes are produced for false positive error reports ("FP" column) compared to fixes for true positives ("Fixes" column).

On the other hand, Infer may find bugs that do not result in a fix (there are typically more "Bugs" than "Fixes" in Table 5.1). FootPatch finds 55 fixes out of 145 bugs (excluding false positives). Patch generation fails when no repair candidate can be found for the bug. Generally, this happens when (a) Infer's analysis times out (e.g., due to loops), leading to incomplete function specifications or short circuited analysis that miss fixing fragments, (b) Infer does not resolve program variables associated with a bug report, (c) no type compatible fragments are discovered, or (d) memory time outs occur for parallel analysis processes, short circuiting analysis results.

Patch location. As motivated in Section 5.3.3, FootPatch heuristically places fix code at the line where Infer reports the violation. For resource and memory leaks, feasible repair locations are constrained by the number of lines at which the resource is no longer in use, but before it is officially dead. In our data, the maximum number of possible correct locations across all 31 fixing patches for resource/memory leaks is 3, while the average is 1.7. This implies little opportunity to vary placement outside of our convention, similar to our motivating example in Figure 5.1a. For null dereferences, checks may plausibly be placed anywhere between the point at which an object becomes null and its dereference. Our inspection based on Infer's bug reports revealed that the number of locations ranges

```
int fd = open(filename, O_RDONLY);
...
swString *content = swString_new(filesize);
if (!content) {
+   close(fd); // FootPatch repair
    return NULL; // function returns with fd not closed
}
... // continues with normal operation
```

Figure 5.9: Resource Leak: forgetting to close a file.

from 1 to 30, which poses more variability for placement.

Patches reveal more bugs. An especially interesting implication of unifying bug detection and repair is the potential for the latter to extend the capabilities of the former. In our experiments, FootPatch generated patches for the `redis` project that then allowed Infer to find two additional unique bugs. Figure 5.8 shows a snippet of the code in question. The `fp` file pointer leaks on at least two paths (lines 6 and 12). Before patching, Infer only reports a resource-leak for the variable `fp`, because it "short circuits" its analysis once the first leak is detected. After FootPatch inserts `fclose(fp);` on line 5, Infer reports the second leak on line 11. Rerunning FootPatch yields an additional fix on line 11. To the best of our knowledge, this is the first demonstration that automated patching has the potential to improve static analysis.

5.4.4 Fixing by Semantic Effects

Resource and Memory Leaks. Resource leaks often occur when a function returns prematurely due to an error [14, 205]. Figure 5.9 shows a leak of an unreleased file handle in the `Swoole`[10] project. FootPatch uses the Repair Query $\{\ \mathit{pvar} \mapsto \langle \mathit{Alloced},\ \mathit{File} \rangle\ \}\ ?C\ \{\ \mathit{pvar} \mapsto \langle \mathit{Freed},\ \mathit{File} \rangle\ \}$ from (6), Figure 5.7, to discover a `close(fd);` elsewhere in the program. This demonstrates the importance of the compatibility check, which guards against applying alternative "close" operations (e.g., `fclose`) by using typing information. The pull request

[10] `Swoole` is the 34th most popular C project on GitHub at the time of writing.

based on the patch in Figure 5.9 was accepted,[11] an important milestone for *end-to-end* automatic repair of a previously undiscovered bug.

Note that, although conceptually similar to resource leaks, memory leaks deserve separate semantic treatment because they tend to occur in programs written in languages that are not garbage collected. Anecdotally, memory leaks may entail more complex fixes in terms of semantic effects. All resource leak patches conform to strict Repair Specifications (meaning they only affect the heap location of interest). However, fixing fragments for memory leaks may entail extra semantic effects. One example fragment is `swHashMap_node_free(hmap, root);` from the `Swoole` project, which frees a data member `root` that is in the table `hmap`. A necessary precondition to inserting this fragment for freeing `root` is that `hmap` be in scope (which it is, where it is used in our produced patches). We obtain such a fragment by relaxing the Repair Query to allow extra semantic effects. One implication of relaxing repairs is that application may be contextual, and subject to additional compatibility checks (e.g., scope and variable capture) with respect to extra semantic effects. In summary, our results show that the majority of leak fixes conform to strict Repair Specifications, but relaxing the repair constraint enables more complex fixes.

Null dereferences. The FootPatch patch for the null dereference(s) in Figure 5.2 throws an exception when an object is null. However, in general, multiple semantic fix effects may address a given null dereference: initializing a null object, returning or throwing an exception when an object is null, or predicating execution on a condition that an object is not null. More than one of these forms may be correct with respect to preserving a non-null property, and in general we cannot decide which one is preferred [143].

We therefore experimented with Repair Specification queries in FootPatch over multiple SIL commands to discover null dereference fixes (i.e., function calls entailing nullness checks, conditional expressions). Our approach alleviates the problem of multiple

[11] https://github.com/swoole/swoole-src/commit/e12c7db38c9737234695d35d9

potential fixes by relying on existing code to guide repair. For example, for `jfreechart`, FootPatch produces 22 patches from a candidate which throws an exception when the object is null. This may be the desirable fix, as witnessed by human-written fixes for the `error-prone` bug [9]. Regardless, from a semantic perspective, FootPatch finds candidate fixes from the existing project that removes the fault with respect to the analysis.

5.5 Limitations and Discussion

FootPatch currently fixes resource leaks, memory leaks, and null dereferences; these bugs are a mature focus of Infer's analysis domain and are common in practice [59, 61]. Given both our underlying technique, which addresses general heap-based properties, and the continual addition of new analyses to Infer [12], we expect FootPatch to generalize to, for example, information flow bugs. FootPatch currently requires a simple manual fix specification, generally per bug type. This formulation provides flexibility to address particular attributes over diverse bug classes and languages. The manual effort is competitive with effort required to produce a test per bug, required by dynamic repair techniques. Moreover, we recognize an opportunity for automatically inferring fixing effects with the aid of a static analyzer, for future work.

FootPatch does not consider the full diversity of possible fixes for the considered defects, especially for null dereferences (we investigated checking nullness and throwing an exception, accepted ways for fixing these bugs, but not instantiating new objects generally). Our approach currently inspects only semantic patch characteristics, ignoring, for example, string constants in an error message. We leave such considerations to future work.

Beyond false positives, Infer can produce inaccurate fault locations, impacting the validity of FootPatch patches. Approximate locations are acceptable for bug reports, and approximate patches may still be informative [204], but truly automated program repair requires precise locations. This can be addressed practically by improving the accuracy of

IL-to-source translation during analysis.

We do not explicitly compare our technique to prior repair techniques. Overall, FootPatch is orthogonal (and thus difficult to compare) to prior dynamic repair techniques. Two of our bug classes, resource and memory leaks, are difficult to test for deterministically, and thus underaddressed in the current repair literature. There does exist work addressing repair of null dereferences, which are easier to expose via tests. We attempted to run and fix the null dereferences in Apache Commons Math considered by NOPOL [211]. However, Infer's analysis skips a number of intermediate calls, and fails to detect the null dereferences covered in the associated test suite. Practically speaking, we find that tests for null dereferences considered in prior work (such as Defects4J [118]) simply cover different null dereference bugs from Infer. This highlights one property of static analysis for repair: analysis may miss bugs that could be covered by tests, but may simultaneously find those in corner cases that humans neglected to test. Related techniques based on verification [143] lack detailed breakdown of bugs to inform a comparative study, and is intended to provide patch suggestions to a developer, which lacks automatic patch application. We are unaware of "quick fix" suggestions from existing static tools [9, 10, 116] that target semantic bugs like those we consider.

5.6 Related Work

Work in automatic program repair over the past decade predominantly use test cases to validate correctness. Generate-and-validate or heuristic repair techniques explore search spaces of templated repairs as applied to the abstract syntax tree program level; this includes techniques like GenProg [135], RSRepair [180], and HDRepair [133] which traverse the space using classic search algorithms like genetic programming or random search. Other techniques like AE [207] SPR [146] and Prophet [145] use predefined transformation schemas and probabilistic models on the AST to discover and apply candidate syntactic fragments.

At a high level, such techniques operate on ASTs to indirectly achieve a desired semantic effect that fixes a bug with respect to a test suite. Constraint- or semantics-based approaches reason about semantics more directly, synthesizing fragments using input-output pairs to codify a notion of program semantics [153, 154, 164, 211]. SearchRepair [120] lies between these approaches, using input-output pairs to search over a semantic encoding of candidate repair fragments (and is perhaps closest in spirit to our approach). These techniques all share the property that they use test cases to define patch correctness and guide a search towards a semantically-desirable fix; as a result, they also require developer labor to specify those tests; are limited to fixing bugs that are deterministically testable, and may overfit to the provided tests [190]. The main point of contrast with our work is that our approach is static, and therefore cannot overfit to provided dynamic witnesses of desired behavior. In the previous body of work, properties are not specified formally, but implied by test cases. FootPatch instead considers a logic-encoding and semantic implications of using a program fragment for satisfying repair specification. We argue that our focus on explicitly codifying semantic effects offers additional protection against patch overfitting. Gao et al. address memory leak fixes for C by inserting a `free` statement [92]. In contrast, our approach identifies fixing fragments based on inferred semantic traits. Although we rely on function models, our approach does not assume any semantic effects of a function call, like `free`. This allows FootPatch to generalize to multiple languages and also identify project-specific fixes (like `free` wrapper functions).

Verification-based approaches, using formal specifications, have used LTL specifications [115, 202], SAT approaches [97], deductive synthesis [126], contracts [143], and model checking for Boolean programs [100, 188] to perform repair. At a high level, our approach relates to the approach by Logozzo et al. [143] which uses automatically inferred assertions over abstract domains, and relies on an abstract interpretation-based static analyzer to discover faults; however, they do not consider automatically applying patches. In contrast, our approach is new in reasoning over an abstract domain based in separation logic (repair-

ing pointer safety violations) and formalizes a mechanism for automatic patch application. Overall, verification-based program repair lack application to common bugs in real-world programs. Adjacent to our work, recent techniques in program synthesis use separation logic encodings to generate heap-manipulating programs from program fragments [179].

5.7 Summary

We presented a new static APR technique using separation logic to reason about semantic effects of program fragments, including a novel application of local reasoning and the frame rule to find bug-fixing patches from existing code. Our technique overcomes significant challenges compared to test-based repair techniques, including the ability to repair previously undiscovered bugs, bugs that are difficult to expose via testing, and repeated semantic errors. We implemented our approach in a tool called FootPatch that builds on top of a separation logic-based analysis to target bug repair for heap-related properties. We demonstrated our approach on resource leaks, memory leaks, and null dereferences (we elaborate on possible extensions in Section 4.9). FootPatch fixes 55 bugs, including 11 undiscovered bugs in 11 projects. Moreover, FootPatch achieves significant speedup over test-based repair, works on large codebases, and targets multiple source languages. Unlike other formal approaches for program repair, FootPatch works end-to-end on existing code bases and does not require formal annotation or special coding practices. Its reliance on principled semantic reasoning provides additional evidence of generated patch correctness. FootPatch thus represents an important step in bridging the gap between grounded verification techniques and trustworthy automatic program repair for real-world software, opening promising avenues in automatic program improvement. We envision direct application of future APR techniques for improving static analyses (as discussed in Section 5.4.3) and auxiliary applications like Semantic Crash Bucketing that benefit from ground truth fixes (Chapter 4).

Chapter 6

Conclusion and Future Work

6.1 Summary

This dissertation focused on the thesis that automated search and application of program transformations enables efficient, scalable, and unassisted techniques for improving the effectiveness of existing program analyses and repair of real-world programs. The premise of the work is that automated program transformation affords new ways to overcome limitations in existing techniques to improve software quality. We focused predominantly on domains that continue to incur substantial human effort despite the relative success of relevant automated techniques. In particular: discerning true and false positives in static analysis, discerning true bug reports versus duplicate reports in dynamic fuzz testing, and correctly fixing bugs.

Developing new techniques where program transformation is a first order concern requires (in whole or in part) a flexible way to generally manipulate syntax. Our solution is a multi-language approach to declarative program transformation using match and rewrite templates, which we presented in Chapter 2.

In Chapter 3 we developed a method for tailoring programs (in particular, using declarative templates enabled by Chapter 2) with the objective to improve static analysis

fidelity. We showed that tailoring programs with selective transformations *can* improve analysis by reducing the number of false positives reported without adversely affecting analysis behavior. One particularly appealing aspect of our solution is that it presents new ways for addressing long-standing, challenging, and unresolved issues in static analyzers (Section 3.2).

In Chapter 4 we introduced a new method for deduplicating bug reports as a function of program transformation. We proposed to automatically generate *approximate* fixes that simulate true developer fixes to discriminate the set of (potentially duplicate) crashing input variants discovered during fuzz testing. We incorporated semantic predicates (using dynamic trace information) and syntax transformation (using Chapter 2) to drive an automated approach for bug class specific approximate fixes. The main result showed that approximate fixes are sufficiently accurate to precisely discriminate duplicate crashes compared to true developer fixes, and outperforms generic heuristics in current deduplication methods.

Finally, we presented a new automated program repair approach for correctly fixing bugs by extending an existing static analysis (Chapter 5). This approach stands in contrast to the predominant methods in automated repair approaches which rely on tests to detect or validate bug fixes, and are therefore limited to fixing only known bugs. A primary benefit of our approach is that we leverage existing automated reasoning for defect detection in the analyzer to fix previously unknown bugs. Moreover, we show that the domain of analysis (i.e., separation logic) enables a new way to search and apply semantically correct fixes.

Our work demonstrates pertinent interrelated concerns that influence the effectiveness of automated techniques for software quality. For example, we saw that our automated repair approach (Chapter 5) depends on the precision of the underlying analysis—if the analysis produces a false positive report, then we may produce a patch that "repairs" that false positive. However, we showed that we can tailor programs to remove false positives with transformation (Chapter 3). Thus, program tailoring not only improves defect detection, but also presents potential for improving the effectiveness of automated program repair. In

the other direction, we have evidence that our repair technique (Chapter 5) can surface and fix more true positive bugs after an initial fix is applied. Thus, automated repair (i.e., achieved by using deeper semantic reasoning) presents further opportunity to amplify the effectiveness of static analyses and bug detection. Lastly, the continuing development of mature and precise automated repair techniques for various bug classes could provision *actual* fixes in place of approximate fixes (as developed in Chapter 4) to broadly improve dynamic fuzz testing in finding and classifying unique bugs.

The techniques in this dissertation would have limited merit if they could not be shown to work effectively on real programs. We therefore evaluated on large, popular programs to demonstrate that all of our techniques *scale* to the demand of actively developed software. We showed that these techniques are *effective* in each domain: relative to existing techniques and approaches, we improved in our ability to easily change programs (Chapter 2), reduced analyzer false positives (Chapter 3), reduced duplicate bug reports (Chapter 4), and repaired previously unknown bugs (Chapter 5). We showed that our techniques are unassisted, modulo up-front specification. Up-front specification is generally easy (via declarative templates; Chapters 2 and 3), or low-effort (i.e., specifying semantic traits that are general per bug class; Chapters 4 and 5). Our evaluation substantiates the claims that these techniques work effectively on large, popular programs and present the prospect of real-world adoption.

6.2 Open Questions and Research Pursuits

We close with open questions and ongoing research pursuits for the work presented in this dissertation.

6.2.1 Extending Automated Program Repair to Additional Bug Classes

Extending our flavor of automated repair in Chapter 5 to handle additional bug classes presents a natural and compelling case for current industry needs. Our intuition is that semantic search and application using pre- and postcondition effects extends generally to bug classes like use-after-free bugs, integer or buffer overflows bugs, and information flow bugs. While separation logic is fit for heap-properties, we expect that additional bug classes will impose the use of alternative abstract domains [83]. For example, repairing information flow bugs would likely entail a taint model, and fixing fragments may concern the introduction or removal of taint flows.

An outstanding challenge is that effective static analyses (like those considered in Chapter 3) are not typically implemented with program repair in mind. As a concrete example, we showed that searching over semantic effects of individual program fragments finds desirable fixes that conform to existing project conventions (Chapter 5). However, the intermediate states of individual program fragments are usually discarded by an analysis. From a bug-finding perspective, intermediate states tend to become irrelevant once a violation is found, or once the effects of a function are summarized. Strikingly, these intermediate states provide key data for reasoning incrementally and compositionally about *changing* parts of programs. Our experience adapting an existing analysis revealed that reasoning and manipulation of intermediate state is scattered across the implementation, making the task difficult and brittle. Program repair *as a first class concern* thus imposes new considerations for designing and optimizing analysis domains and implementations. Our position is that program repair, building on static analysis, will truly thrive when interfaces for exposing intermediate state representation and reasoning become a first order concern.

6.2.2 Cooperative Automated Program Repair and Program Analysis

We showed that targeted program transformations improves static and dynamic analyses (Chapters 3 and 4). This can in turn improve the fidelity of automated repair. In the reverse direction, automated program repair can reveal new bugs in static analyses, or substitute real fixes for approximate ones in dynamic settings (Chapters 5). Our work broadly reveals the potential for cooperative interaction between automated program repair, program transformation, and analysis for the first time. We envision that as automated program repair matures, it will find increasing application in directing and improving analyses. Much remains to be explored in the space of cooperative repair and program tailoring in tandem with symbolic execution [42, 58, 157], runtime verification [103, 148], testing [93] and testability transformation [107, 108].

6.2.3 Inference in Place of Up-front Specification

The techniques in this dissertation generally start with varying forms of up-front specification to initiate an automated process. Syntactic transformation and program tailoring entail the specification of declarative match templates (Chapters 2 and 3). Semantic Crash Bucketing (Chapter 4) uses declarative templates and semantic predicates using dynamic traces to inform patch application. FootPatch (Chapter 5) relies on small specifications per bug class that express desirable semantic pre- and postconditions to identify candidate patches. Automatically *inferring* these specifications presents an appealing direction to increase efficiency and generality to additional bugs. At a high level, ease and utility of inference is subject to the complexity of the technique and bug class in question. For example, causal relationships between program semantics and analysis behavior can be inferred by speculatively adding or removing program fragments (e.g., via counterfactual reasoning [65]). At a syntactic level, we may infer changes from historic commits that work around static

analysis issues (as in Section 3.1) to better tailor programs. However, inference towards fully autonomic procedures is not always desirable or applicable. Concretely, program tailoring for static analysis may entail program transformations that are only desirable as a temporary suppression mechanism, and not for persisting in code (Section 3.4.3). Similarly, allowing a mechanism where developers and security practitioners can integrate their specialist knowledge into an automated process (like fuzzing and crash deduplication in Section 4.1) is a desirable trait that remains attractive in the absence of inferred properties. Ultimately, inferring automated transformation presents a complementary approach to the techniques discussed.

6.2.4 Automation and Integration with Developer Workflows

Automation is a key trait for increasing developer efficiency and software quality techniques. However, automated processes are not necessarily fully autonomous: surrounding an automated process are humans who regularly configure, deploy, and monitor the process. As discussed in Chapter 3, developers are sensitive to whether tools can integrate into existing workflows [116]. Thus, automated techniques must integrate with existing software processes like continuous integration and testing to be effective [152]. Facilitating this integration for large scale program transformation and repair is an open problem. One appealing direction is the use of automated agents: software bots that bridge the gap between human software development and automated processes. Our position is that repair bots (e.g., [83, 195] present new ways for orchestrating the unique capabilities that come with automated refactoring and repair. We share these thoughts in greater detail in a separate publication of ours [198].

6.2.5 Enriching Declarative Transformation with Semantic Information

Automating code changes that require semantic information (e.g., types) is difficult, and especially so at scale [210]. Such sophisticated transformations are usually implemented in complex frameworks by experts who have deep knowledge of a language's representation and tooling (e.g., `Clang` for C/C++). Our goal with `Comby` is to make program transformation simple and flexible, as an alternative to understanding underlying language ASTs and visitor frameworks. At the time of writing, we are in the process of extending `Comby` to make use of the rich semantic information available to language specific tools (like types, scoping, and def-use relationships) so that sophisticated transformations are more accessible across multiple languages. As a concrete example, this extension allows us to use type information in the presence of syntactic ambiguity to discern wrong transformations (as in Section 2.5.1 for Go).

Our approach relies on recent efforts using the Language Server Protocol (LSP) [33]. The LSP standardization makes it possible to query semantic information (e.g., types) for a project hosted in a central location (where compilation and semantic information is cached). Such queries are thus efficient and generalize to multiple languages. We see LSP as a promising development towards decoupling static semantics from program syntax and locally managed compilation. While we do not aim to reach feature parity with existing language-specific tooling (like `Clang`), there is broad appeal in delivering a declarative method of program transformation using semantic attributes to developers at large.

There is increasing uptake and a pressing industry need for automating code changes, from automated dependency updates [30] to API migrations [210]. We anticipate that improving the accessibility, sophistication, and efficiency of the automated program transformation techniques in this dissertation rises to the challenge of current and future needs.

Bibliography

[1] Clang Static Analyzer. https://clang-analyzer.llvm.org/. Accessed 4 May 2019.

[2] CodeSonar. https://www.grammatech.com/products/codesonar. Accessed 4 May 2019.

[3] Coverity: suppressing asserts. https://community.synopsys.com/s/question/0D534000046YuzbCAC. Accessed 4 May 2019.

[4] Error Prone: Patching. https://errorprone.info/docs/patching. Accessed 4 May 2019.

[5] Infer. https://github.com/facebook/infer. Accessed 4 May 2019.

[6] NullAway: auto-suppressing. https://github.com/uber/NullAway/wiki/Suppressing-Warnings#auto-suppressing. Accessed 4 May 2019.

[7] PHPStan. https://github.com/phpstan/phpstan. Accessed 4 May 2019.

[8] Spotbugs. https://github.com/spotbugs/spotbugs. Accessed 4 May 2019.

[9] Google Error-prone bug-fixing commit. https://github.com/google/error-prone/commit/3709338, 2017. Online; accessed 16 January 2017.

[10] FindBugs Static Analyzer. https://github.com/findbugsproject/findbugs, 2017. Online; accessed 26 August 2017.

[11] Infer bug types. http://fbinfer.com/docs/infer-bug-types.html, 2017. Online; accessed 11 May 2017.

[12] Infer experimental checkers. http://fbinfer.com/docs/experimental-checkers.html, 2017. Online; accessed 11 May 2017.

[13] Infer Static Analyzer. http://fbinfer.com/, 2017. Online; accessed 11 May 2017.

[14] Resource Leak in C. http://fbinfer.com/docs/infer-bug-types.html#RESOURCE_LEAK, 2017. Online; accessed 16 January 2017.

[15] https://lcamtuf.blogspot.com/2015/04/finding-bugs-in-sqlite-easy-way.html, 2018. Online; accessed 26 April, 2018.

[16] AFL-Fuzz. http://lcamtuf.coredump.cx/afl/, 2018. Online; accessed 26 April, 2018.

[17] Public vulnerabilities discovered using bff. https://vuls.cert.org/confluence/display/tools/Public+Vulnerabilities+Discovered+Using+BFF, 2018. Online; accessed 26 April, 2019.

[18] CERT BFF. https://www.cert.org/vulnerability-analysis/tools/bff-download.cfm, 2018. Online; accessed 26 April, 2018.

[19] https://cve.mitre.org/, 2018. Online; accessed 26 April, 2018.

[20] Rewrite Transforms in Facebook's Hack. https://github.com/facebook/hhvm/blob/master/hphp/hack/src/parser/coroutine/coroutine_state_machine_generator.ml#L413-L433, 2018. Online; accessed 16 March 2018.

[21] Honggfuzz. https://github.com/google/honggfuzz, 2018. Online; accessed 26 April, 2018.

[22] CVE-2017-12762. https://patchwork.kernel.org/patch/9880041/, 2018. Online; accessed 26 April, 2018.

[23] OSS-Fuzz. https://github.com/google/oss-fuzz, 2018. Online; accessed 26 April 2018.

[24] https://access.redhat.com/security/security-updates/#/cve, 2018. Online;

accessed 26 April, 2018.

[25] Microsoft Security Risk Detection. https://www.microsoft.com/en-us/security-risk-detection/, 2018. Online; accessed 26 April, 2018.

[26] Clang's refactoring engine. https://clang.llvm.org/docs/RefactoringEngine.html, Online. Accessed 16 April 2019.

[27] Facebook's Codemod for large-scale codebase refactors. https://github.com/facebook/codemod, Online. Accessed 16 April 2019.

[28] The Effective Dart style guide. https://www.dartlang.org/guides/language/effective-dart/style, Online. Accessed 16 April 2019.

[29] decaffeinate. https://github.com/decaffeinate/decaffeinate, Online. Accessed 16 April 2019.

[30] Dependabot: Automated dependency updates. https://dependabot.com/, Online. Accessed 16 April 2019.

[31] gofmt Go code formatter. https://golang.org/cmd/gofmt, Online. Accessed 16 April 2019.

[32] Instaparse. https://github.com/Engelberg/instaparse, Online. Accessed 16 April 2019.

[33] The Language Server Protocol. https://langserver.org, Online. Accessed 16 April 2019.

[34] Pylint: a Python linter. https://media.readthedocs.org/pdf/pylint/latest/pylint.pdf, Online. Accessed 16 April 2019.

[35] Refal: REcursive Functions Algorithmic Language. http://www.refal.net/~arklimov/refal6/, Online. Accessed 16 April 2019.

[36] Rosie Pattern Language. https://developer.ibm.com/code/open/projects/rosie-pattern-language/, Online. Accessed 16 April 2019.

[37] staticcheck for Go. http://staticcheck.io/docs/#overview, Online. Accessed 16 April 2019.

[38] TSLint: a linter for TypeScript. https://palantir.github.io/tslint/rules/, Online. Accessed 16 April 2019.

[39] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98, 2007.

[40] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. Generating focused random tests using directed swarm testing. In *International Symposium on Software Testing and Analysis*, ISSTA '16, pages 70–81, 2016.

[41] Rajeev Alur and P. Madhusudan. Visibly Pushdown Languages. In *Symposium on Theory of Computing*, pages 202–211, 2004.

[42] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing Symbolic Execution with Veritesting. In *International Conference on Software Engineering*, ICSE '14, pages 1083–1094, 2014.

[43] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Experiences Using Static Analysis to Find Bugs. *IEEE Software*, 25: 22–29, 2008. Special issue on software development tools.

[44] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[45] Jonathan Bachrach and Keith Playford. D-expressions: Lisp power, dylan style. 1999.

[46] Jonathan Bachrach and Keith Playford. The Java Syntactic Extender. In *Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '01, pages

31–42, 2001.

[47] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods*, IFM '04, pages 1–20. Springer, 2004.

[48] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson R. Murphy-Hill. From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration. In *IEEE International Conference on Software Maintenance and Evolution*, ICSME '16, pages 211–221. IEEE Computer Society, 2016.

[49] J Berdine, C Calcagno, and Peter W O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*, FMCO '05, pages 115–137, 2005.

[50] Josh Berdine, Cristiano Calcagno, and Peter W O'hearn. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages and Systems*, APLAS '05, pages 52–68, 2005.

[51] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Verification Condition Generation and Variable Conditions in Smallfoot. *CoRR*, abs/1204.4804, 2012.

[52] Josh Berdine, Arlen Cox, Samin Ishtiaq, and Christoph M. Wintersteiger. Diagnosing Abstraction Failure for Separation Logic-Based Analyses. In *Computer Aided Verification*, CAV '12, pages 155–173, 2012.

[53] Jean Berstel and Luc Boasson. Balanced Grammars and Their Languages. In *Formal and Natural Computing* , pages 3–25, 2002.

[54] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, February 2010.

[55] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Sci. Comput. Program.*, 72(1-2):52–70, 2008.

[56] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible Debugging Software. Technical report, University of Cambridge, Judge Business School, 2013.

[57] Fraser Brown, Andres Nötzli, and Dawson R. Engler. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 143–157, 2016.

[58] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Operating Systems Design and Implementation*, OSDI '08, pages 209–224, 2008.

[59] Cristiano Calcagno and Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*, NFM '11, pages 459–465, 2011.

[60] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

[61] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving Fast with Software Verification. In *NASA Formal Methods*, NFM '15, pages 3–11, 2015.

[62] Jacques Chabin and Pierre Réty. Visibly Pushdown Languages and Term Rewriting. In *Frontiers of Combining Systems*, pages 252–266, 2007.

[63] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. Angelic Debug-

ging. In *International Conference on Software Engineering*, ICSE '11, pages 121–130, 2011.

[64] Swarat Chaudhuri and Rajeev Alur. Instrumenting C Programs with Nested Word Monitors. In *International SPIN Symposium on Model Checking of Software*, pages 279–283, 2007.

[65] Sheng Chen and Martin Erwig. Counter-factual Typing for Debugging Type Errors. In *Principles of Programming Languages*, POPL '14, pages 583–594, 2014.

[66] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming Compiler Fuzzers. In *Conference on Programming Language Design and Implementation*, PLDI '13, pages 197–208, 2013.

[67] Maria Christakis and Christian Bird. What Developers Want and Need from Program Analysis: An Empirical Study. In *International Conference on Automated Software Engineering*, ASE '16, pages 332–343, 2016.

[68] Maria Christakis, Peter Müller, and Valentin Wüstholz. An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer. In *Verification, Model Checking, and Abstract Interpretation*, VMCAI '15, pages 336–354, 2015.

[69] Holger Cleve and Andreas Zeller. Locating Causes of Program Failures. In *International Conference on Software Engineering*, ICSE '05, pages 342–351, 2005.

[70] Zack Coker and Munawar Hafiz. Program Transformations to Fix C Integers. In *International Conference on Software Engineering*, ICSE '13, pages 792–801, 2013.

[71] Michael L. Collard and Jonathan I. Maletic. srcML 1.0: Explore, Analyze, and Manipulate Source Code. In *International Conference on Software Maintenance and Evolution*, ICSME '16, page 649, 2016.

[72] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *International Working*

Conference on Source Code Analysis and Manipulation, SCAM '11, pages 173–184, 2011.

[73] James R. Cordy. The TXL Source Transformation Language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[74] J. Robert M. Cornish, Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Analyzing Array Manipulating Programs by Program Transformation. In *Logic-Based Program Synthesis and Transformation*, LOPSTR '14, pages 3–20, 2014.

[75] Patrick Cousot and Radhia Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Symposium on Principles of Programming Languages*, POPL '02, pages 178–190, 2002.

[76] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *International Conference on Software Engineering*, ICSE '16, pages 820–831, 2016.

[77] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing Static Analyzers with Randomly Generated Programs. In *NASA Formal Methods*, NFM '12, pages 120–125, 2012.

[78] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity. In *International Conference on Software Engineering*, ICSE '12, pages 1084–1093, 2012.

[79] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification*, CAV '16, pages 383–401, 2016.

[80] Dino Distefano and Ivana Filipovic. Memory Leaks Detection in Java by Bi-abductive Inference. In *Fundamental Approaches to Software Engineering*, FASE, pages 278–292, 2010.

[81] Dino Distefano and Matthew J Parkinson J. jStar: Towards Practical Verification for Java. *ACM Sigplan Notices*, 43(10):213–226, 2008.

[82] Dino Distefano, Peter W O'Hearn, and Hongseok Yang. A Local Shape Analysis Based on Separation Logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302, 2006.

[83] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling Static Analyses at Facebook. *Commun. ACM*, 62(8):62–70, July 2019. ISSN 0001-0782.

[84] Walter J Doherty and Arvind J Thadhani. The Economic Value of Rapid Response Time. *IBM Report*, 1982.

[85] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *PACMPL*, 1(OOPSLA):93:1–93:29, 2017.

[86] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1992.

[87] Pär Emanuelsson and Ulf Nilsson. A Comparative Study of Industrial Static Analysis Tools. *Electr. Notes Theor. Comput. Sci.*, 217:5–21, 2008.

[88] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Software*

Language Engineering, SLE '13, pages 197–217, 2013.

[89] Eldar Fischer, Frédéric Magniez, and Tatiana A. Starikovskaya. Improved Bounds for Testing Dyck Languages. In *Symposium on Discrete Algorithms*, SODA '18, pages 1529–1544, 2018.

[90] Bryan Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, 2004.

[91] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer Overrun Detection using Linear Programming and Static Analysis. In *Conference on Computer and Communications Security*, CCS '03, pages 345–354, 2003.

[92] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. Safe Memory-Leak Fixing for C Programs. In *International Conference on Software Engineering*, ICSE '15, pages 459–470, 2015.

[93] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. Crash-Avoiding Program Repair. In *International Symposium on Software Testing and Analysis*, ISSTA '19, 2019.

[94] Seymour Ginsburg and Michael A. Harrison. Bracketed Context-Free Languages. *J. Comput. Syst. Sci.*, 1(1):1–23, 1967.

[95] Patrice Godefroid and Daniel Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *International Symposium on Software Testing and Analysis*, ISSTA '11, page 23, 2011.

[96] Denis Gopan, Evan Driscoll, Ducson Nguyen, Dimitri Naydich, Alexey Loginov, and David Melski. Data-Delineation in Software Binaries and its Application to Buffer-Overrun Discovery. In *International Conference on Software Engineering*, ICSE '15, pages 145–155, 2015.

[97] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-Based Program Repair Using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '11, pages 173–188, 2011.

[98] Rahul Gopinath, Carlos Jensen, and Alex Groce. The Theory of Composite Faults. In *International Conference on Software Testing, Verification*, ICST '17, pages 47–57, 2017.

[99] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. A True Positives Theorem for a Static Race Detector. *PACMPL*, 3(POPL):57:1–57:29, 2019.

[100] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of Boolean Programs with an Application to C. pages 358–371, 2006.

[101] Robert Grimm. Better Extensibility through Modular Syntax. In *Programming Language Design and Implementation*, PLDI '06, pages 38–51, 2006.

[102] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm Testing. In *International Symposium on Software Testing and Analysis*, ISSTA '12, pages 78–88, 2012.

[103] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Rosu. RV-Match: Practical Semantics-Based Program Analysis. In *Computer Aided Verification*, CAV '16, pages 447–453. Springer, 2016.

[104] Andrew Habib and Michael Pradel. How Many of all Bugs do we Find? A Study of Static Bug Detectors. In *Automated Software Engineering*, ASE '18, pages 317–328, 2018.

[105] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular Checking for Buffer Overflows in the Large. In *International Conference on Software Engineering*, ICSE '06, pages 232–241, 2006.

[106] Mark Harman. Automated Patching Techniques: The Fix is In: Technical perspective.

Commun. ACM, 53(5):108, 2010.

[107] Mark Harman. We Need a Testability Transformation Semantics. In *Software Engineering and Formal Methods*, SEFM '18, pages 3–17, 2018.

[108] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability Transformation. *IEEE Trans. Software Eng.*, 30(1):3–16, 2004.

[109] Christoph M Hoffmann and Michael J O'Donnell. Pattern Matching in Trees. *Journal of the ACM*, 29(1):68–95, 1982.

[110] Gerard J. Holzmann. Cobra: a Light-weight Tool for Static and Dynamic Program Analysis. 13(1):35–49, 2017.

[111] Gerard J. Holzmann. Cobra: fast structural code checking (keynote). In *International SPIN Symposium on Model Checking of Software*, pages 1–8, 2017.

[112] Graham Hutton. Higher-Order Functions for Parsing. *Journal of Functional Programmming*, 2(3):323–343, 1992.

[113] Graham Hutton and Erik Meijer. Monadic Parser Combinators, 1996.

[114] Ciera Jaspan, I-Chin Chen, and Anoop Sharma. Understanding the Value of Program Analysis Tools. In *Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 963–970, 2007.

[115] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program Repair as a Game. In *Computer Aided Verification*, CAV '05, pages 226–238, 2005.

[116] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *International Conference on Software Engineering*, ICSE '13, pages 672–681, 2013.

[117] James A. Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *International Conference on Automated*

Software Engineering, ASE '05, pages 273–282, 2005.

[118] René Just, Darioush Jalali, and Michael D Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *International Symposium on Software Testing and Analysis*, ISSTA '14, pages 437–440, 2014.

[119] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Interactive Disambiguation of Meta Programs with Concrete Object Syntax. In *Software Language Engineering*, pages 327–336, 2010.

[120] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing Programs with Semantic Code Search. In *International Conference on Automated Software Engineering*, ASE '15, pages 295–306, 2015.

[121] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic Patch Generation Learned from Human-written Patches. In *International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[122] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic Patch Generation Learned from Human-written Patches. In *International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[123] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A Field Study of Refactoring Challenges and Benefits. In *Foundations of Software Engineering*, FSE '12, page 50, 2012.

[124] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *Computer and Communications Security*, CCS '18.

[125] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177, 2009.

[126] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive Program Repair. In *Computer Aided Verification*, CAV '15, pages 217–233, 2015.

[127] James Koppel, Varot Premtoon, and Armando Solar-Lezama. One Tool, Many Languages: Language-Parametric Transformation with Incremental Parametric Syntax. *PACMPL*, 2(OOPSLA):122:1–122:28, 2018.

[128] Shuvendu K. Lahiri, Rohit Sinha, and Chris Hawblitzel. Automatic Rootcausing for Program Equivalence Failures in Binaries. In *Computer Aided Verification*, CAV '15, pages 362–379, 2015.

[129] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, dec 1992.

[130] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *USENIX Security Symposium*, 2001.

[131] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization*, CGO '04, pages 75–88, 2004.

[132] Julia Lawall and Gilles Muller. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *USENIX Annual Technical Conference*, pages 601–614, 2018.

[133] Xuan-Bach D. Le, David Lo, and Claire Le Goues. History Driven Program Repair. In *Software Analysis, Evolution, and Reengineering*, SANER '16, pages 213–224, 2016.

[134] Xuan Bach D Le, David Lo, and Claire Le Goues. History Driven Program Repair. In *Software Analysis, Evolution, and Reengineering*, SANER '16, pages 213–224, 2016.

[135] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *International Conference on Software Engineering*, ICSE '12, pages 3–13, 2012.

[136] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current Challenges in Automatic Software Repair. *Software Quality Journal*, 21(3):421–443, 2013.

[137] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report, Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[138] Josh Levenberg. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM*, 59(7):78–87, 2016.

[139] Frank Li and Vern Paxson. A Large-Scale Empirical Study of Security Patches. In *Conference on Computer and Communications Security*, CCS '17, pages 2201–2215, 2017.

[140] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable Statistical Bug Isolation. In *Programming Language Design and Implementation*, PLDI '05, pages 15–26, 2005.

[141] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair. In *Symposium on Information, Computer and Communications Security*, pages 329–340, 2007.

[142] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. *Commun. ACM*, 58(2):44–46, 2015.

[143] Francesco Logozzo and Thomas Ball. Modular and Verified Automatic Program Repair. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '12, pages 133–146, 2012.

[144] Francesco Logozzo and Manuel Fähndrich. On the Relative Completeness of Bytecode

Analysis Versus Source Code Analysis. In *Compiler Construction*, CC '08, pages 197–212, 2008.

[145] Fan Long and Martin Rinard. Automatic Patch Generation by Learning Correct Code. In *Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, 2016.

[146] Fan Long and Martin C. Rinard. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *International Conference on Software Engineering*, ICSE '16, pages 702–713, 2016.

[147] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *Conference on Programming Language Design and Implementation*, PLDI '14, pages 227–238, 2014.

[148] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian-Florin Serbanuta, and Grigore Rosu. RV-Monitor: Efficient Parametric Runtime Verification with Simultaneous Properties. In *RV*, volume 8734 of *Lecture Notes in Computer Science*, pages 285–300. Springer, 2014.

[149] Laura MacLeod, Michaela Greiler, Margaret-Anne D. Storey, Christian Bird, and Jacek Czerwonka. Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Software*, 35(4):34–42, 2018.

[150] Jonathan I. Maletic and Michael L. Collard. Exploration, Analysis, and Manipulation of Source Code Using srcML. In *International Conference on Software Engineering*, ICSE '15, pages 951–952, 2015.

[151] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. SapFix: Automated End-to-end Repair at Scale. In *International Conference on Software Engineering: Software Engineering in Practice*, SEIP@ICSE '19, pages 269–278. IEEE, 2019.

[152] Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. Test Activities in the Continuous Integration and Delivery Pipeline. *Journal of Software: Evolution and Process*, 31(4), 2019.

[153] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *International Conference on Software Engineering*, ICSE '15, pages 448–458, 2015.

[154] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *International Conference on Software Engineering*, ICSE '16, pages 691–701, 2016.

[155] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Trans. Software Eng.*, 44(5):453–469, 2018.

[156] Eduardus A. T. Merks, J. Michael Dyck, and Robert D. Cameron. Language Design For Program Manipulation. *IEEE Trans. Software Eng.*, 18(1):19–32, 1992.

[157] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *USENIX Security Symposium*, pages 67–82, 2009.

[158] Martin Monperrus. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, 2018.

[159] Leon Moonen. Generating Robust Parsers Using Island Grammars. In *Conference on Reverse Engineering.*

[160] Paul Muntean, Vasantha Kommanapalli, Andreas Ibing, and Claudia Eckert. Automated Generation of Buffer Overflow Quick Fixes Using Symbolic Execution and SMT. In *Computer Safety, Reliability, and Security*, SAFECOMP '15, pages 441–456,

2015.

[161] Nachiappan Nagappan and Thomas Ball. Use of Relative Code Churn Measures to Predict System Defect Density. In *International Conference on Software Engineering*, ICSE ’05, pages 284–292, 2005.

[162] Kedar S. Namjoshi and Zvonimir Pavlinovic. The Impact of Program Transformations on Static Program Analysis. In *International Symposium on Static Analysis*, SAS ’18, pages 306–325, 2018.

[163] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. *Electronic notes in Theoretical Computer Science*, 89(2):44–66, 2003.

[164] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program Repair via Semantic Analysis. In *International Conference on Software Engineering*, ICSE ’13, pages 772–781, 2013.

[165] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, VMCAI ’07, pages 251–266, 2007.

[166] Jakob Nielsen. *Usability engineering*. Elsevier, 1994.

[167] National Institute of Standards and Technology. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical Report NIST Planning Report 02-3, NIST, 2002. URL http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[168] Peter O’Hearn. Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[169] Peter W. O’Hearn. From Categorical Logic to Facebook Engineering. In *Symposium on Logic in Computer Science*, pages 17–20, 2015.

[170] Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about

Programs that Alter Data Structures. In *International Workshop on Computer Science Logic*, CSL '01, pages 1–19, 2001.

[171] Chris Okasaki. Functional Pearl: Even Higher-Order Functions for Parsing. *J. Funct. Program.*, 8(2):195–199, 1998.

[172] Joonyoung Park, Inho Lim, and Sukyoung Ryu. Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild. In *International Conference on Software Engineering*, ICSE '16, pages 61–70, 2016.

[173] Matthew J. Parkinson and Gavin M. Bierman. Separation Logic and Abstraction. In *Symposium on Principles of Programming Languages*, POPL '05, pages 247–258, 2005.

[174] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. Quoted Staged Rewriting: A Practical Approach to Library-defined Optimizations. In *International Conference on Generative Programming: Concepts and Experiences*, GPCE '17, pages 131–145, 2017.

[175] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

[176] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy*, 2018.

[177] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically Patching Errors in Deployed Software. In *Symposium on Operating Systems Principles*, SIGOPS '09, pages 87–102, 2009.

[178] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. Bucketing Failing Tests via Symbolic Analysis. In *Fundamental Approaches to Software*

Engineering Conference, FASE '17, pages 43–59, 2017.

[179] Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *PACMPL*, 3(POPL):72:1–72:30, 2019.

[180] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The Strength of Random Search on Automated Program Repair. In *International Conference on Software Engineering*, ICSE, pages 254–265, 2014.

[181] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *International Symposium on Software Testing and Analysis*, ISSTA '15, pages 24–36, 2015.

[182] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *USENIX Security Symposium*, pages 861–875, 2014.

[183] Manos Renieris and Steven P. Reiss. Fault Localization with Nearest Neighbor Queries. In *International Conference on Automated Software Engineering*, ASE '03, pages 30–39, 2003.

[184] Martin C Rinard, Cristian Cadar, Daniel Dumitran, Daniel M Roy, Tudor Leu, and William S Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Operating Systems Design and Implementation*, volume 4 of *OSDI '04*, pages 303–316, 2004.

[185] Sukyoung Ryu. Scalable Framework for Parsing: From Fortress to JavaScript. *Softw., Pract. Exper.*, 46(9):1219–1238, 2016.

[186] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from Building Static Analysis Tools at Google. *Commun. ACM*, 61 (4):58–66, 2018.

[187] Konstantinos Sagonas and Thanassis Avgerinos. Automatic Refactoring of Erlang Programs. In *International Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 13–24, 2009.

[188] Roopsha Samanta, Oswaldo Olivo, and E Allen Emerson. Cost-Aware Automatic Program Repair. In *Static Analysis Symposium*, SAS '14, pages 268–284, 2014.

[189] Kostya Serebryany. OSS-Fuzz–Google's Continuous Fuzzing Service for Open Source Software. In *USENIX Security Symposium*, 2017.

[190] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, pages 532–543, 2015.

[191] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. A Deeper Look Into Bug Fixes: Patterns, Replacements, Deletions, and Additions. In *International Conference on Mining Software Repositories*, MSR '16, pages 512–515, 2016.

[192] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do Software Engineers Understand Code Changes?: An Exploratory Study in Industry. In *Foundations of Software Engineering*, FSE '12, page 51, 2012.

[193] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the Localness of Software. In *Foundations of Software Engineering*, FSE 2014, pages 269–280, 2014.

[194] Valentin F. Turchin. The Concept of a Supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

[195] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. How to Design a Program Repair Bot?: Insights from the Repairnator Project. In *International Conference on Software Engineering: Software Engineering in Practice*, pages 95–104,

2018.

[196] Rijnard van Tonder and Claire Le Goues. Tailoring Programs for Static Analysis via Program Transformation *(under submission)*, 2019.

[197] Rijnard van Tonder and Claire Le Goues. Static Automated Program Repair for Heap Properties. In *International Conference on Software Engineering*, ICSE '18, pages 151–162, 2018.

[198] Rijnard van Tonder and Claire Le Goues. Towards s/engineer/bot: Principles for Program Repair Bots. In *Bots in Software Engineering*, BotSE@ICSE '19, pages 43–47, 2019.

[199] Rijnard van Tonder and Claire Le Goues. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Programming Language Design and Implementation*, PLDI '19, pages 363–378, 2019.

[200] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. Semantic Crash Bucketing. In *Automated Software Engineering*, ASE '18, pages 612–622, 2018.

[201] Eelco Visser. Meta-programming with Concrete Object Syntax. In *Generative Programming and Component Engineering*, GPCE '02, pages 299–315, 2002.

[202] Christian von Essen and Barbara Jobstmann. Program Repair without Regret. *Formal Methods in System Design*, 47(1):26–50, 2015.

[203] Louis Wasserman. Scalable, Example-based Refactorings with Refaster. In *Workshop on Refactoring Tools*, WRT@SPLASH '13, pages 25–28, 2013.

[204] Westley Weimer. Patches As Better Bug Reports. In *Generative Programming and Component Engineering*, GPCE '06, pages 181–190, 2006.

[205] Westley Weimer and George C. Necula. Mining Temporal Specifications for Error Detection. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '05, pages 461–476, 2005.

[206] Westley Weimer and George C. Necula. Exceptional Situations and Program Reliability. *ACM Transactions on Programming Languages and Systems*, 30(2):8:1–8:51, March 2008.

[207] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *Automated Software Engineering*, ASE '13, pages 356–366, 2013.

[208] Daniel Weise and Roger F. Crew. Programmable Syntax Macros. In *Programming Language Design and Implementation*, PLDI '93, pages 156–165, 1993.

[209] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling Black-box Mutational Fuzzing. In *Conference on Computer & Communications Security*, CCS '13, pages 511–522, 2013.

[210] Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. Large-Scale Automated Refactoring Using ClangMR. In *International Conference on Software Maintenance*, ICSM '13, pages 548–551, 2013.

[211] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.*, 43(1):34–55, 2017.

[212] Hongseok Yang and Peter O'Hearn. A Semantic Basis for Local Reasoning. In *International Conference on Foundations of Software Science and Computation Structures*, FoSSaCS, pages 402–416, 2002.

[213] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Programming Language Design and Implementation*, PLDI '11, pages 283–294, 2011.

[214] Vadim Zaytsev. Formal Foundations for Semi-parsing. In *IEEE Conference on*

Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE '14, pages 313–317, 2014.

[215] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.