Automatic Patch Generation via Learning from Successful Human Patches

by

Fan Long

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
October 20, 2017

Signature redacted

Certified by.......
Martin C. Rinard
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by ....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Automatic Patch Generation via Learning from Successful Human Patches

by

Fan Long

Submitted to the Department of Electrical Engineering and Computer Science
on October 20, 2017, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Automatic patch generation holds out the promise of automatically correcting software defects without the need for developers to manually diagnose, understand, and correct these defects. This dissertation presents two novel patch generation systems, Prophet and Genesis, which learn from past successful human patches to enhance the patch generation process. The core of Prophet and Genesis is a novel learning technique that extracts universal properties of correct code and a novel inference technique that generalizes universal patching strategies across different applications. The results show that the learning and inference techniques enable Prophet and Genesis to operate with rich and tractable search spaces that contain many useful patches and efficient search algorithms that prioritize correct patches. By collectively leveraging development efforts worldwide, Prophet and Genesis automatically generate correct patches for real-world defects in large open-source C and Java applications with up to millions lines of code.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

It is really hard to start here. Time flies. I have lived in Boston for seven years and now it is the time to say goodbye to everyone. Seven years are really long. I feel almost everything in my life has changed. Seven years are also really short. I still remember the day when I first came to MIT for the open house visit and that day feels like yesterday. During my seven years at MIT CSAIL lab, I was fortunate to work with most brilliant people in the planet. I want to say thank to all those people who helped me during the last seven years.

First of all, I want to thank my advisor Martin Rinard. Working with Martin is a life-time unique experience. He is always optimistic, encouraging, energetic, and full of crazy ideas. Although I do not always agree with him, every conversation with him is fun and interactive. When I look back, I feel that choosing him as my PhD advisor was one of the best decisions I ever made in my life. I learned so many things from him including writing, presentation, and most importantly how to think like a computer scientist. Under his supervision, I have grown up from a young hacker to a mature researcher and thinker.

I also received many help and advice from other faculty members in and outside MIT CSAIL, especially Regina Barzilay, Armondo Solar-Lezama, Arvind, and Suresh Jagannathan. They gave me help and feedback on my research and they provide me valuable support when I was on the job market.

I want to thank all the people in our group, especially Stelios Sidiroglou-Douskos, Michael Carbin, Deokhwan Kim, Sasa Misailovic, Jeff Perkins, Michael Gordon, Sara Achour, Zichao Qi, Jiasi Shen, and Eric Lahtinen. As a group, we shared many fun times and deadline nights together, which I will definitely miss in future. We had many lounge conversations which later turned into great research projects and papers.

During my seven years, I lived together with three great roommates Tao Lei, Haogang Chen, and Rui Li. We often eat together, play together, discuss homework and research together. I was especially fortunate to have great collaborations with Tao on an inter-field project. I was fortunate to meet many other friends in MIT

CSAIL outside our group as well, especially Xi Wang, Peng Wang, Yuan Zhang, Cai Yang, and Even Pu. Xi and I had many research collaborations even before I came to MIT and he inspired me to become a computer science researcher.

I also want to thank my family. My parents always support me with their unconditional love during my hardest time. Without them this thesis would not be possible. Finally, I want to thank my wife Yu Lao. Our love is always the largest force in my heart that drives me moving forward.

Contents

1 Introduction 21

1.1 Generate-and-Validate Systems 22

1.2 Learning from Successful Human Patches 23

1.3 Learning Universal Correctness Properties 24

1.4 Learning Universal Patching Strategies 26

1.5 Usage Scenario 28

1.6 Contributions 29

1.7 Roadmap 31

2 Analysis of Patch Plausibility and Correctness 33

2.1 Research Questions 33

2.2 Overview of Analyzed Systems 38

2.3 Plausibility Analysis 40

2.4 Correctness Analysis 44

2.5 GenProg Reexecutions 50

2.6 Semantic Patch Analysis 51

2.6.1 Weak Test Suites 52

2.6.2 Case Studies 52

2.6.3 Impact of Functionality Deletion Patches 62

2.7 Original GenProg Patches 64

2.8 Kali 68

2.8.1 Kali Evaluation Methodology 69

2.8.2 Experimental Results 70

2.9 Discussion 75

3 A General Framework for Generate-and-Validate Systems 79

3.1 Run Defect Localiation 80

3.2 Apply Transforms 81

3.3 Rank Candidate Patches 82

3.4 Validate with Test Cases 83

3.5 Offline Learning in Prophet and Genesis 85

4 Learning Universal Properties of Correct Code with Prophet 87

4.1 Motivating Example 89

4.2 Transformation Schemas and Condition Synthesis in Prophet and SPR 94

4.2.1 Core Language 95

4.2.2 Transformation Schemas 97

4.2.3 Validation Algorithm with Condition Synthesis 99

4.2.4 SPR Patch Validation Order 104

4.3 Learning Universal Properties 106

4.3.1 Probabilistic Model 106

4.3.2 Defect Localization Approximation for Learning 108

4.3.3 Learning Algorithm 109

4.3.4 Feature Extraction 110

4.3.5 Feature Extraction for Abstract Conditions 116

4.3.6 Repair Algorithm 119

4.3.7 Alternative Learning Objective 119

4.4 Implementation 121

4.4.1 Defect Localization 121

4.4.2 C Program Support and Optimizations 123

5 Evaluation of Prophet Patch Generation System 125

5.1 Benchmarks 126

5.2 Human Patch Collection and Training 127

5.2.1 Collect Successful Human Patches 127

5.2.2 Train Prophet on Collected Training Set 128

5.3 Patch Generation Evaluation 129

5.3.1 Methodology 129

5.3.2 Experimental Results 130

5.3.3 Case Studies 132

5.4 Design Decision Evaluation 139

5.4.1 Methodology 139

5.4.2 Comparison of Different Variant Systems 140

5.4.3 Per-Defect Results of Prophet and SPR 142

5.4.4 Condition Synthesis 145

5.5 Search Space Extensions and Evaluation 146

5.5.1 Methodology 146

5.5.2 Search Space Analysis 147

5.5.3 Defects Outside Prophet Search Space 148

5.5.4 Search Space Extensions 149

5.5.5 Results of Search Space Extentions 151

5.5.6 Plausible and Correct Patch Density 155

5.5.7 Search Space Tradeoffs 156

5.5.8 Prophet and SPR Effectiveness 159

5.6 Discussion 159

5.6.1 Prophet Hypothesis 160

5.6.2 Important Features 160

5.6.3 Search Space Tradeoff 161

6 Learning Universal Patching Strategies with Genesis 163

6.1 Motivating Example 166

6.2 Inference Algorithm 170

6.2.1 Preliminaries 170

6.2.2 Template AST Forest, Generator, Transforms 172

6.2.3 Transform Generalization 175
6.2.4 Sampling Algorithm 180
6.2.5 Search Space Inference Algorithm 180
6.3 Implementation 182
6.3.1 Integration with Patch Characteristic Learning 182
6.3.2 Integration with Condition Synthesis 184
6.3.3 Defect Localization 185
6.3.4 Handling Java Programs 186

7 Evaluation of Genesis Patch Generation System 189
7.1 Benchmark and Training Defects 190
7.2 Offline Inference and Learning 191
7.3 Patch Generation 192
7.3.1 Methodology 192
7.3.2 Experimental Results 193
7.3.3 Case Studies 195
7.4 Genesis Transforms 199
7.4.1 Transform Overview 199
7.4.2 Comparison with PAR 202
7.4.3 Comparison with Prophet 203
7.4.4 Defects Outside Genesis Search Space 203
7.5 Patch Prioritization and Condition Synthesis 204
7.5.1 Methodology 204
7.5.2 Experimental Results 205
7.6 Discussion 206
7.6.1 Comparison of Genesis and PAR 206
7.6.2 Patch Prioritization in Genesis and Prophet 207
7.6.3 Condition Synthesis in Genesis and Prophet 208

8 Future Directions and Realistic Expectations 211
8.1 Future Directions 211

8.2 Realistic Expectations and Limitations 213

9 Related Work 215

9.1 Automatic Patch Generation 215

9.1.1 Generate-and-validate Techniques 216

9.1.2 Leveraging Human Patches 220

9.1.3 Specification-based Techniques 222

9.1.4 Targeted Techniques 223

9.1.5 Feedback Generation for Programming Education 225

9.2 Learning from Existing Programs 226

9.3 Defect Localization 228

9.4 Code Refactoring 229

9.5 Discussion 230

10 Conclusion 231

A Prophet and SPR Results Per Search Space Configuration 233

B Genesis Per Defect Experimental Results 267

List of Figures

2-1 GenProg Patch for python-bug-69783-69784. Delete Lines 1-49. . . . 45

2-2 Developer Patch for python-bug-69783-69784. Delete Lines 1-27. . . . 46

2-3 GenProg Patch for php-bug-309892-309910. Delete Lines 15-17. . . . 48

2-4 Developer Patch for php-bug-309892-309910. Delete Lines 8-10. . . . 48

2-5 AE Patch for php-bug-309111-309159. Insert lines 5-15. 49

2-6 Developer Patch for php-bug-309111-309159. Insert lines 2-6. 50

2-7 Developer Patch for libtiff-bug-0860361d-1ba75257. Modify Lines 9-10. 53

2-8 GenProg and AE Patches for libtiff-bug-0860361d-1ba75257. Delete Lines 8-10. 53

2-9 Developer Patch for php-bug-307931-307934. Insert Line 7. 54

2-10 GenProg and AE Patches for php-bug-307931-307934. Delete Lines 11-12. 55

2-11 Developer Patch for php-bug-310673-310681. Insert Lines 9-30 and Lines 35-38. 57

2-12 GenProg and AE Patches for php-bug-310673-310681. Delete Lines 11-13. 58

2-13 Developer Patch for gzip-bug-3fe0ca-39a362. Modify Lines 10-11, Lines 14-15, and Line 19. 58

2-14 GenProg Patch for gzip-bug-3fe0ca-39a362. Insert Line 3 and Lines 14-15. 59

2-15 Developer Patch for gzip-bug-a1d3d4-f17cbd. Insert Line 6. 61

2-16 AE Patch for gzip-bug-a1d3d4-f17cbd. Insert Line 17. 61

2-17 Kali Patch for python-bug-69783-69784. Modify Lines 1-2. 73

2-18 Kali Patch for php-bug-309892-309910. Modify Lines 1-2. 74

2-19 Kali Patch for php-bug-311346-311348. Modify lines 8-9. 74

2-20 Developer Patch for php-bug-311346-311348. Modify Lines 1-2. . . . 75

3-1 The Patch Generation Workflow of Generate-and-Validate Systems 80

3-2 The Offline Learning of Prophet and Genesis 85

4-1 Simplified Code for PHP bug #53971 91

4-2 The Language for Illustrating Transformation Schemas 94

4-3 Small Step Operational Semantics for Statements without Abstract Condition 96

4-4 The Program Transformation Function M 98

4-5 Patch Validation Algorithm with Condition Synthesis 100

4-6 Definitions of Exec, Test, Flip, and F 101

4-7 Small Step Operational Semantics for if-statements with Abstract Condition 102

4-8 Learning Algorithm 109

4-9 The Language for Illustrating Feature Extraction 111

4-10 Definitions and Notation. **SK** corresponds to the set of statement kinds. **MK** corresponds to the set of modification kinds. **AC** corresponds to the set of atomic characteristics that the analysis function ψ extracts. 111

4-11 Feature Extraction Algorithm 112

4-12 Atomic Characteristic Extraction Rules for $\psi(p, a, n)$ 113

4-13 Prophet Repair Algorithm 120

5-1 Prophet Patch for php-308262-308315. Add Line 11. 133

5-2 Prophet Patch for gzip-a1d3d4-f17cbd. Add Line 3. 133

5-3 SPR Patch for gzip-a1d3d4-f17cbd. Add Line 2. 134

5-4 Prophet Patch for libtiff-d13be-ccadf. Add Line 3. 134

5-5 SPR Patch for libtiff-d13be-ccadf. Add Line 3. 135

5-6 Prophet Patch for fbc-5458-5459. Add Line 19. 136

5-7 SPR Patch for fbc-5458-5459. Add Line 10. 137

5-8 A Plausible But Incorrect Patch Generated by Prophet for php-310011-310050. Modify Lines 4-5. 137

5-9 Developer Patch for php-310011-310050. Add Line 4. 138

5-10 Developer Patch for php-307563-307571. Modify Lines 1-16. 138

6-1 Example Inference and Application of a Genesis Transform. The training patches (original and patched code) are at the top, the inferred transform is in the middle, and the new patch that Genesis generates is at the bottom. 167

6-2 Example of Using Integer Linear Programming to Select an Effective Set of Transforms. 167

6-3 Definitions of $\models$ and $\models_{\texttt{slice}}$ 174

6-4 Definition of the $\Longrightarrow$ for the Generator $\mathcal{G} = \langle G, b, \delta, W\rangle$ 174

6-5 Definition of $\Longrightarrow$ and $\Longrightarrow_{\texttt{slice}}$ for the Transform $\mathcal{P}$ 176

6-6 Definition of the Generator Inference Operator ψ 177

6-7 Definition of the Generalization Function Ψ 177

6-8 Definition of Ψ' 178

6-9 Sampling Algorithm sample(D, E) 181

6-10 Integer Linear Programming Formulas for Selecting Transforms Given a Set of Candidate Transforms $\mathbb{P}'$ and a Validation Set of AST Slice Pairs E 181

6-11 The Condition Synthesis Algorithm in Genesis 184

7-1 Genesis Correct Patch for spring-data-rest-aa28aeb. Add Lines 5-7. . 195

7-2 Genesis Correct Patch for jPOS-df400ac. Add Lines 3-5. 196

7-3 Genesis Correct Patch for jade4j-114e886. Modify Lines 6-7. 197

7-4 Developer Patch for nutz-80e85d0. Add Lines 2 and 4-5. 197

7-5 Developer Patch for commons-lang-52b46e7. Modify Lines 1 and 9-16. 198

List of Tables

2.1 Kali Experimental Results 67

4.1 Atomic Characteristics of Program Values for C 117

5.1 Benchmarks for Prophet Evaluation 126

5.2 Statistics of Collected Successful Human Patches 128

5.3 The Number of Correct Patches Per Application Per System 130

5.4 The Number of Plausible Patches Per Application Per System 131

5.5 Comparative Results for Different Systems 140

5.6 Prophet Per-defect Patch Generation Results 143

5.7 SPR Per-defect Patch Generation Results 144

5.8 Modification Type of Prophet Correct Patches 147

5.9 Search Space Extensions 150

5.10 Patch Generation Results with Search Space Extensions (excluding php) 152

5.11 Patch Generation Results with Search Space Extensions (php only) . 153

5.12 Costs and Payoffs of Reviewing the First 10 Generated Patches (excluding php) 158

5.13 Costs and Payoffs of Reviewing the First 10 Generated Patches (php only) 158

6.1 Atomic Characteristics of Program Values for Java in Genesis 183

7.1 Genesis Search Space Inference Results 191

7.2 Genesis Patch Generation Results 193

7.3 Patch Generation Results for Three Different Versions of Genesis. . . 205

A.1 Prophet-100 Statistics 234

A.2 Prophet-100-CExt Statistics 235

A.3 Prophet-100-RExt Statistics 236

A.4 Prophet-100-RExt-CExt Statistics 237

A.5 Prophet-200 Statistics 238

A.6 Prophet-200-CExt Statistics 239

A.7 Prophet-200-RExt Statistics 240

A.8 Prophet-200-RExt-CExt Statistics 241

A.9 Prophet-300 Statistics 242

A.10 Prophet-300-CExt Statistics 243

A.11 Prophet-300-RExt Statistics 244

A.12 Prophet-300-RExt-CExt Statistics 245

A.13 Prophet-2000 Statistics 246

A.14 Prophet-2000-CExt Statistics 247

A.15 Prophet-2000-RExt Statistics 248

A.16 Prophet-2000-RExt-CExt Statistics 249

A.17 SPR-100 Statistics 250

A.18 SPR-100-CExt Statistics 251

A.19 SPR-100-RExt Statistics 252

A.20 SPR-100-RExt-CExt Statistics 253

A.21 SPR-200 Statistics 254

A.22 SPR-200-CExt Statistics 255

A.23 SPR-200-RExt Statistics 256

A.24 SPR-200-RExt-CExt Statistics 257

A.25 SPR-300 Statistics 258

A.26 SPR-300-CExt Statistics 259

A.27 SPR-300-RExt Statistics 260

A.28 SPR-300-RExt-CExt Statistics 261

A.29 SPR-2000 Statistics 262

A.30 SPR-2000-CExt Statistics 263

A.31 SPR-2000-RExt Statistics 264

A.32 SPR-2000-RExt-CExt Statistics 265

B.1 Results of the Genesis Combined Search Space on NP Defects 269

B.2 Results of the Genesis Combined Search Space on OOB Defects . . . 270

B.3 Results of the Genesis Combined Search Space on CC Defects 271

B.4 Results of the Genesis Per-Defect-Type Search Space on NP Defects . 272

B.5 Results of the Genesis Per-Defect-Type Search Space on OOB Defects 273

B.6 Results of the Genesis Per-Defect-Type Search Space on CC Defects . 274

B.7 Results of the Genesis Combined Search Space with Condition Synthesis on NP Defects 275

B.8 Results of the Genesis Combined Search Space with Condition Synthesis on OOB Defects 276

B.9 Results of the Genesis Combined Search Space with Condition Synthesis on CC Defects 277

B.10 Results of the Genesis Combined Search Space without Patch Prioritization Learning on NP Defects 278

B.11 Results of the Genesis Combined Search Space without Patch Prioritization Learning on OOB Defects 279

B.12 Results of the Genesis Combined Search Space without Patch Prioritization Learning on CC Defects 280

B.13 Results of PAR Templates on NP Defects 281

B.14 Results of PAR Templates on OOB Defects 282

B.15 Results of PAR Templates on CC Defects 283

Chapter 1

Introduction

Software applications are integrated into every part of our society. There are more software programs than ever before, e.g., the number of open source repositories in GitHub reached 60 million at 2017, and this number continues to grow [16]. As the dependence of our society on these programs also continues to grow, making these programs reliable and secure becomes an increasingly important challenge for our society and a daunting task for software developers. Automated techniques for improving software reliability and security will be even more important than ever before.

Software defects are pervasive in software systems and can cause undesirable user experience, denial of service, or even security exploitation [89, 108]. Generating a patch for a defect is a tedious, time-consuming, and often repetitive process. Human developers have to diagnose the defect from the collected bug reports, understand its root cause, craft the patch to correct the defect, and validate the patch with regression tests and code reviews [108, 109]. Even for defects that cause security vulnerabilities, it takes 28 days on average for maintainers to develop and distribute fixes [13]. Out of the 1.7 million GitHub issues about Java runtime exceptions at the time of writing this dissertation, more than half million issues are still open. This demonstrates that we are running out of development resources to fix these issues and defects in time.

Automatic patch generation [30, 49, 55, 65, 73, 79, 80, 82, 104] holds out the promise of automatically correcting software defects without the need for human developers to diagnose, understand, and correct these defects. This dissertation presents novel patch generation techniques that learn from past successful human patches. The techniques treat the large volume of existing programs not just a challenge but also an opportunity. This opportunity enables the techniques to leverage past development efforts to enhance the patch generation process. Specifically, this dissertation presents the first technique that learns universal properties of correct code to recognize correct patches. This dissertation also presents the first technique that learns universal patching strategies across different applications to generate rich and tractable search spaces of candidate patches.

1.1 Generate-and-Validate Systems

Automatic patch generation techniques can be broadly classified into three categories: 1) targeted techniques that repair specific classes of universal defects such as null dereferences [61], out of bounds accesses [90], and infinite loops [28, 50], 2) synthesis-based techniques that leverage formal specifications to produce patches that enable a defective program to satisfy the specification [30, 79], and 3) *generate-and-validate techniques*, which work with a test suite of inputs, generate a set of candidate patches, then test the patched programs against the test suite to find a patch that validates [49, 55, 65, 73, 80, 82, 104]. This dissertation focuses on generate-and-validate techniques.

The standard generate-and-validate approach starts with a test suite of inputs, at least one of which exposes a defect in the software [55, 82, 104]. The system then generates a space of candidate patches and searches this space to find *plausible patches* that produce correct outputs for all inputs in the test suite. The advantage of generate-and-validate systems is that they can apply to general classes of defects without the need of formal specifications, which are typically not available for large real world applications.

Analysis of Previous Systems: To give a backdrop of previous generate-and-validate systems, Chapter 2 in this dissertation systematically analyzes patch generation results from three previous systems, GenProg [55], AE [104], and RSRepair [82].

The analysis substantially changed the field's understanding of the capabilities of the three analyzed systems. The overwhelming majority of the reported patches are *incorrect* and many of them are semantically equivalent to eliminating existing functionalities in the original program. For the majority of the evaluated benchmark defects, all generated patches of GenProg, AE, and RSRepair are incorrect, because the search space of the systems contains no correct patch.

Plausible but Incorrect Patches: Our analysis reveals one important challenge of generate-and-validate systems: *plausible but incorrect patches* caused by *weak test suites*. Because the supplied test suites typically do not fully capture the desired behavior of the program, the patch generation systems often accept plausible but incorrect patches that produce correct results for all of the inputs in the test suite but incorrect outputs for other inputs. These plausible but incorrect patches often have significant negative effects such as eliminating desired functionality or introducing security vulnerabilities.

Search Space Tradeoff: Our analysis, together with the evaluation of our own generate-and-validate system in Chapter 5, reveals another important challenge of generate-and-validate systems: the inherent trade-off between *the coverage and tractability* of the candidate patch search space. For a patch generation system to generate a correct patch for a defect, on one hand, the search space of the system has to contain the correct patch. On the other hand, the search space has to be small enough so that the patch generation system can efficiently explore the space to find the correct patch. All previous patch generation systems rely on manually defined code transform rules to generate candidate patches and these manual rules may have suboptimal trade-offs – they may miss many correct patches or generate too many candidate patches which cause the search space explosion.

1.2 Learning from Successful Human Patches

Our analysis results highlight the importance of exploiting additional information other than test suites to improve the search space and the search algorithm in generate-

and-validate systems. One way to obtain such information is from past successful human patches (e.g., collected from open source repositories).

This dissertation presents learning and inference techniques that extract and exploit useful information from a set of training human patches. These techniques enable patch generation systems to leverage development efforts worldwide to automatically fix defects. A key observation is that standard automatic patch generation systems operate in a completely different way than human developers. A generate-and-validate patch generation system exhaustively enumerates all candidate patches in its search space with its superior computation power. Human developers often unconsciously apply certain software engineering strategies to consider only those patches that exhibit certain properties of successful code. The ultimate goal of my research is to build advanced patch generation systems that combine the computation power of machines with the sophisticated insights of human developers.

This dissertation also presents the design, implementation, and evaluation of two novel patch generation systems, Prophet and Genesis, which implement the new learning and inference techniques. Prophet and Genesis automatically learn universal properties of successful human patches in a database and use the learned information to prioritize potentially correct patches in their search space. Genesis further automatically learns universal patching strategies from the human patches to infer a productive search space of candidate patches instead of relying on manually defined code transform rules. Our evaluation results show that Prophet and Genesis automatically generate correct patches for real-world defects in large open-source C and Java applications with up to millions lines of code.

1.3 Learning Universal Correctness Properties

Chapter 4 presents Prophet, the first generate-and-validate patch generation system that learns from past successful human patches. Prophet works for C programs. It first learns a probabilistic model of correct code from the training patches. The learned model assigns a probability score to each candidate patch in the Prophet search space.

Prophet then uses the model to rank candidate patches in order of likely correctness, prioritizing potentially correct patches among many candidate patches in the search space.

A key challenge for Prophet is to identify and learn *universal properties of correct code*. Each patch inserts new code into the program. But correctness does not depend only on the new code — it also depends on how that new code interacts with the program into which it is inserted. Prophet therefore extracts not only the characteristics of the patch itself, but also interactions between the patch and surrounding code. Prophet appropriately abstracts away syntactic details such as variable names to produce a set of application-independent features.

Experimental Results: Chapter 5 evaluates Prophet on 69 real world defects drawn from eight large open source applications. Our results show that, on the same benchmark set, Prophet outperforms previous generate-and-validate patch generation systems, specifically GenProg [55] and AE [104]. Within its 12 hour time limit, Prophet finds correct patches for 18 of these 19 defects. For 15 of these 19 defects, the first patch to validate is correct. GenProg and AE find correct patches for 1 and 2 defects, respectively.

The results also show that the learned patch search order is critical to the success of Prophet. A baseline system that uses a random order to prioritize its search of the same space of candidate patches, finds correct patches for only 14 of these 19 defects (four less than Prophet). For only 7 of these 19 defects (eight less than Prophet), the first patch to validate is correct. SPR, a patch generation system which uses a set of of hand-coded heuristics to prioritize its search of the same space, finds correct patches for 16 of these 19 defects (two less than Prophet). For 11 of these 19 defects (four less than Prophet), the first patch to validate is correct.

By designing and evaluating Prophet, this dissertation therefore investigates the following hypothesis:

Hypothesis. Correct patches across different applications share universal properties that can be learned from past successful human patches.

Our results are consistent with this hypothesis – by learning and exploiting universal properties of correct code, Prophet explores its search space more efficiently and therefore generates more correct patches than previous systems. Notably, Prophet learns universal properties of successful human patches from one set of applications and applies the learned information to generate correct patches for defects in a different application.

1.4 Learning Universal Patching Strategies

Chapter 6 presents Genesis, the first patch generation system that automatically learns universal patching strategies from successful human patches to infer code transforms and the search space. Genesis works for Java programs. Unlike all previous patch generation systems, Genesis does not rely on any manually crafted transform. It instead uses the automatically inferred transforms to generate candidate patches. These patches embody a rich variety of different patching strategies developed by a wide range of human developers, and not just the patch generation strategies encoded in a set of manually crafted transforms from the developers of the patch generation system.

Genesis captures the diverse patch generation strategies with its novel representation of *code transforms*, each of which is composed of two *template abstract syntax trees* (AST) to capture the common code structures before and after applying the transform. One template AST matches code in the original program; the other template AST specifies the replacement code for the generated patch. Template ASTs contains *template variables*, which match subtrees in the original or patched code. Template variables enable the transforms to abstract away application specific details to capture common patterns implemented by multiple patches drawn from different applications.

Many useful patches do not simply rearrange existing code and logic; they also introduce new code and logic. Genesis transforms therefore implement *partial pattern matching* in which the replacement template AST contains free template variables that are not matched in the original code. Each of the free template variables is associated with a *generator*, which systematically generates new candidate code components for the free variable. This new technique, which enables Genesis to synthesize new code and logic in the candidate patches, is essential to enabling Genesis to generate correct patches for previously unseen applications.

Genesis uses a novel generalization algorithm to obtain candidate code transforms from sampled subsets of training human patches. Genesis then selects a subset of candidate transforms as the final inferred search space. Genesis navigates the search space design trade-off between the coverage and tractability by formulating and solving an integer linear program (ILP). The solution of the ILP maximizes the number of training patches covered by the selected transforms while acceptably bounding the number of candidate patches that the selected transforms can generate.

Experimental Results: Chapter 7 presents the evaluation of Genesis. We use Genesis to infer patch search spaces and generate patches for three classes of defects in Java programs: null pointer (NP), out of bounds (OOB), and class cast (CC) defects. Working with a training set that includes 483 NP patches, 199 OOB patches, and 287 CC patches drawn from 356 open source applications, Genesis infers a search space generated by 108 transforms.

Our benchmark defects include 20 NP, 13 OOB, and 16 CC defects from 41 open source applications. All of the benchmark applications are systematically collected from github [16] and contain up to 235K lines of code. With the 108 inferred transforms, Genesis generates correct patches for 21 out of the 49 defects (11 NP, 6 OOB, and 4 CC defects). We compare Genesis with PAR [49, 69], a previous patch generation system for Java that works with manually defined patch templates. For the same benchmark set, the PAR templates generate correct patches for 10 fewer defects (specifically, 7 NP and 4 OOB defects).

We attribute these results to the ability of the automated Genesis inference algorithms to navigate complex patch transform trade-offs at scale. Genesis works with hundreds to over a thousand candidate transforms to obtain productive search spaces generated by close to a hundred selected transforms — many more transforms than any previous generate and validate system. For example, the Genesis combined search space for NP, OOB, and CC defects contains 85 transforms. In contrast, PAR contains only nine templates for these defects and Prophet contains only seven transformation schemas. Deploying this many transforms enables Genesis to capture a broad range of patching strategies, with the transforms selected to ensure the overall tractability and coverage of the resulting patch search space.

By designing and evaluating Genesis, this dissertation therefore investigates the following hypothesis:

> **Hypothesis.** Correct patches across different applications share universal patching strategies that can be learned from past successful human patches.

Our results are consistent with this hypothesis – by learning and deploying universal patching strategies, Genesis operates with more productive search spaces and therefore generates significantly more correct patches than previous systems. Notably, Genesis learns universal patching strategies of successful human patches from one set of applications and applies the learned information to generate correct patches for defects in a potentially different application.

1.5 Usage Scenario

Prophet and Genesis take a program that contains a defect and a set of test cases, at least one of which exposes the defect. Prophet and Genesis produce a ranked list of generated patches, all of which pass the supplied test cases. The ranking of the generated patches is determined by the likelihood of the patch correctness in the learned probabilistic model.

While the basic principles and the techniques behind Prophet and Genesis can be applied broadly to a range of important software engineering problems, one motivating scenario is as follows. Whenever a new bug report with a bug triggering input is submitted to a developer, the developer converts the bug report into a new test case to expose the defect. Then instead of manually crafting patches to fix the defect, the developer applies Prophet or Genesis with the new test case together with existing regression test cases to generate patches for the defect. The developer finally reviews the ranked list of the generated patches one by one following the order to select a correct patch to apply.

Genesis has been adopted and productized by a large outsourcing and software maintenance company. It is currently being demonstrated to customers, to an apparently enthusiastic response, and is scheduled to go into production on customer projects in early 2018. Genesis is particularly valuable in this context because the company's business model involves maintaining large production software systems developed by others. The company's engineers are therefore often unfamiliar with the code and starting largely from scratch when attempting to fix a defect. The ability of Genesis to automatically generate correct patches in this situation is therefore quite valuable to this company and their customer base.

1.6 Contributions

This dissertation demonstrates that exploiting additional information learned from human patches is a very promising direction to build powerful generate-and-validate systems that can leverage past development efforts worldwide. This dissertation presents the first results in this area and lays a foundation for further pursuing this direction. Specifically, this dissertation makes the following contributions:

Analysis of Generate-and-validate Systems: This dissertation presents a systematic analysis of the patch generation results of generate-and-validate systems.[1] It shows that the overwhelming majority of the reported GenProg, RSREpair, and AE patches are incorrect and many of them are equivalent to functionality deletions. It identifies 1) the plausible but incorrect patches and 2) the search space design trade-off between the coverage and tractability as two important challenges of generate-and-validate systems. The analysis has changed the evaluation standard of the field. Since the publication of the analysis results [85], it has become a standard practice

[1] Note that the manual analysis of the majority of the generated patches of GenProg, RSRepair, and AE is performed by Zichao Qi and Sara Achour under the supervision of the author of this dissertation.

to evaluate not just the plausibility but also the correctness of patches produced by generate-and-validate systems [53, 58, 59, 64, 65, 101, 106].

Learning Universal Correctness Properties: This dissertation presents a novel approach for learning universal properties of correct code. This approach uses a parameterized discriminative probabilistic model to assign a correctness probability to candidate patches. This correctness probability captures not only properties of the new code present in the candidate patches, but also properties that capture how this new code interacts with the surrounding code into which it is inserted. It also presents an algorithm that learns the model parameters via a training set of successful human patches collected from open-source project repositories.

Learning Universal Patching Strategies: This dissertation presents a novel approach for learning universal patching strategies of successful human patches to infer code transforms and search spaces. It presents novel transforms with template ASTs, template variables, and generators. It also presents a novel patch generalization algorithm that, given a set of training patches, automatically derives such a transform. It further presents a novel search space inference algorithm that formulates the search space design trade-off between the coverage and tractability as solving an integer linear program.

Prophet and Genesis Implementation: This dissertation presents Prophet, the first patch generation system that learns universal properties of correct code, and Genesis, the first patch generation system that learns both universal properties of correct code and universal patching strategies of successful human patches. Our experimental results show that Prophet and Genesis significantly outperform previous patch generation systems for programs in C and Java, respectively. The results are consistent with our hypotheses – both systems exploit learned information from human patches to enhance their ability to generate correct patches.

1.7 Roadmap

The rest of this dissertation is organized as follows. Chapter 2 presents our systematic analysis of previous generate-and-validate systems. If you are interested in the status of previous generate-and-validate techniques and the background of this research, you can read Chapter 2. Chapter 3 presents a general framework for generate-and-validate systems including Prophet and Genesis. It is an important chapter to read for understanding the high-level architecture of Prophet and Genesis.

If you are interested in Prophet, you can read Chapter 4 and Chapter 5. Chapter 4 presents the design of Prophet including the algorithm for learning universal properties. Chapter 5 presents our evaluation of Prophet. If you are interested in Genesis, you can read Chapter 6 and Chapter 7. Chapter 6 presents the design of Genesis including the algorithm for learning universal patching strategies. Chapter 7 presents our evaluation of Genesis. Chapter 8 discusses future directions and realistic expectations. Chapter 9 presents related work and we finally conclude at Chapter 10.

Chapter 2

Analysis of Patch Plausibility and Correctness

Generate-and-validate patch generation systems operate with a test suite of inputs, at least one of which exposes a defect in the software. The systems apply program modifications to generate a space of candidate patches, then search the generated patch space to find *plausible patches* — i.e., patches that produce correct outputs for all inputs in the test suite. To give a backdrop of previous generate-and-validate patch generation techniques and to set the context for the research presented in this dissertation, in this chapter, we analyze GenProg [55], RSRepair [82], and AE [104] systems.

Sources. The previous version of this research presented in this chapter appeared in [85] and [83]. The majority of the manual analysis of the GenProg, AE, and RSRepair generated patches was performed by Zichao Qi and Sara Anchour under the supervision of the author of this dissertation.

2.1 Research Questions

The reported results for these systems are impressive: GenProg is reported to fix 55 of 105 considered bugs [55], RSRepair is reported to fix all 24 of 24 considered bugs (these bugs are a subset of the 55 bugs that GenProg is reported to fix) [82], and AE is reported to fix 54 of the same 105 considered bugs [104].[1]

To better understand the capabilities and potential of these systems, we performed an analysis of the patches that these systems produce. Enabled by the availability of the generated patches and the relevant patch generation and validation infrastructure [1, 5, 7, 8, 10, 12], our analysis was driven by the following research questions:

RQ1: Do the reported GenProg, RSRepair, and AE patches produce correct outputs for the inputs in the test suite used to validate the patch?

Plausibility Analysis and Weak Proxies: The basic principle behind generate-and-validate systems is to only accept *plausible* patches that produce correct outputs for all inputs in the test suite used to validate the patches. Despite this principle, our analysis shows that because of errors in the patch validation infrastructure, many of the reported patches are, in fact, not plausible – they do not produce correct outputs even for the inputs in the test suite used to validate the patch.

For 37 of the 55 defects that GenProg is reported to repair [55], none of the reported patches produce correct outputs for the inputs in the test suite. For 14 of the 24 defects that RSRepair is reported to repair [82], none of the reported patches that we analyze produce correct outputs for the inputs in the test suite. And for 27 of the 54 defects reported in the AE result tar file [1], none of the reported patches produces correct outputs.

Further investigation indicates that *weak proxies* are the source of the error. A weak proxy is an acceptance test that does not check that the patched program produces correct output. It instead checks for a weaker property that may (or may not) indicate correct execution. Specifically, some of the acceptance tests check only if the patched program produces an exit code of 0. If so, they accept the patch (whether the output is correct or not). To cite two specific examples, both GenProg and AE will accept a patch that completely replaces the main program of either php or libtiff with `return 0` as fixing any bug in these two applications. Specifically, both GenProg

[1] Our analysis of the commit logs and applications indicates that 36 of these bugs correspond to deliberate functionality changes, not actual bugs. That is, for 36 of these bugs, there is no actual bug to fix. To simplify the presentation of this chapter, however, we refer to all of the 105 bugs as defects.

and AE will accept the following program as a correct implementation of both php and libtiff:

```
int main() { return 0; }
```

Because of weak proxies, all of these systems violate the underlying basic principle of generate-and-validate patch generation. The result is that the majority of the patches accepted by these systems do not generate correct outputs even for the inputs in the validation test suite. See Section 2.3.

Correctness Analysis: Despite multiple publications that analyze the reported patches and the methods used to generate them [39, 41, 54, 55, 57, 69, 81, 82, 104], we were able to find no systematic patch correctness analysis. We therefore analyzed the remaining plausible patches to determine if they eliminated the defect or not.

RQ2: Are any of the reported GenProg, RSRepair, and AE patches correct?

The overwhelming majority of the patches are not correct. Specifically, GenProg produced a correct patch for only 2 of the 105 considered defects.[2] Similarly, RSRepair produced a correct patch for only 2 of the 24 considered defects. AE produced a correct patch for only 3 of the 105 considered defects. For each of the incorrect patches, we have a test case that exposes the defect in the patch [84]. Because of weak proxies, many of these test cases were already present in the existing test suites. For the remaining plausible but incorrect patches, we developed new test cases that exposed the defects. See Section 2.4.

Stronger Test Suites: One hypothesis is that stronger test suites with additional inputs that provide better coverage would enable these systems to generate more correct patches:

RQ3: Do stronger test suites enable GenProg to produce more correct patches?

To investigate this question, we reran all of the GenProg runs that generated incorrect patches. We used corrected test scripts and enhanced test suites that

[2] We note that the paper discusses only two patches: one of the correct patches for one of these two defects and a patch that is obtained with the aid of user annotations [55].

contained defect-exposing test cases for all of these patches. These reexecutions produced no patches at all. We next discuss two potential explanations for this result.

A necessary prerequisite for the success of any search-based patch generation algorithm is a search space that contains successful patches. One potential explanation is that the GenProg, RSRepair, and AE search spaces do not contain correct patches for these defects. This explanation is consistent with our evaluation results in Section 5.5.2 for Prophet, whose search space contains correct patches for 20 of the 105 defects in the GenProg benchmark set. Only 3 of these correct patches are within the GenProg search space.

Another potential explanation is that these systems do not use a search algorithm that can explore the search space efficiently enough. GenProg's genetic search algorithm uses the number of passed test cases as the fitness function. For most of the defects in the benchmark set, there is only one *negative test case*, which exposes the defect. Therefore even the unpatched program passes all but one of the test cases. With this fitness function, the difference between the fitness of the unpatched code and the fitness of a plausible patch that passes all test cases is only one. There is therefore no smooth gradient for the genetic search to traverse to find a solution. In this situation, genetic search can easily devolve into random search. Indeed, RSRepair (which uses random search) is reported to find patches more quickly and with less trials than GenProg [82]. See Section 2.5.

Functionality Deletion: As we analyzed patch correctness, it became clear that (despite some surface syntactic complexity), the overwhelming majority of the plausible patches were semantically quite simple. Specifically, they were equivalent to a single functionality deletion modification, either the deletion of a single line or block of code or the insertion of a single return or exit statement.

RQ4: How many of the plausible reported GenProg, RSRepair, and AE patches are equivalent to a single functionality deletion modification?

Our analysis indicates that 104 of the 110 plausible GenProg patches, 37 of the 44 plausible RSRepair patches, and 22 of the 27 plausible AE patches are equivalent to a single modification that deletes functionality.

Our analysis also indicates that (in contrast to previously reported results [56]) some of the plausible patches had significant negative effects, including the introduction of new integer and buffer overflow security vulnerabilities and the elimination of standard desirable functionality. These negative effects highlight some of the risks associated with the combination of functionality deletion and generate-and-validate patch generation. See Section 2.6 for the details of our analysis.

Despite their potential for negative effects, functionality deletion patches can be useful in helping developers locate and better understand defects. For obtaining functionality deletion patches for this purpose, we advocate using a system that focuses solely on functionality deletion (as opposed to a system that aspires to create correct patches). Such an approach has at least two advantages. First, it is substantially simpler than approaches that attempt to generate more complex repairs. The search space can therefore be smaller, simpler, and searched more efficiently. Second, focusing solely on functionality deletion can produce simpler, more transparent patches for developers who want to use the patches to help them locate and better understand defects.

Original GenProg: There are two versions of GenProg. The previous analysis results are for the new version of GenProg published in 2012 [55]. The original GenProg system is published in 2009 [40, 103]. Our analysis of the original GenProg system yields similar results. Out of 11 defects evaluated in the two original GenProg papers, the generated patches for 9 defects are incorrect (in some cases because of the use of weak proxies, in other cases because of weak test suites). The patches for 9 of the 11 defects simply delete functionality (removing statements, adding return statements, or adding exit statements). The only two defects for which the original GenProg system generates correct patches are small motivating examples (less than 30 lines of code). See Section 2.7 for the details of our analysis.

ManyBugs Benchmark Set: GenProg, AE, and RSRepair (under the name TRPRepair) have recently been evaluated on a larger set of benchmark defects [54]. Our analysis shows that the ManyBugs test infrastructure still suffers from errors present in the GenProg [55] and AE [104] systems. These errors again stem from the use of weak proxies in the testing infrastructure. The reported ManyBugs results therefore exhibit similar errors as the earlier GenProg [55] and AE [104] results.

Kali: Inspired by the observation that the patches for the vast majority of the defects that GenProg, RSRepair, and AE were able to address consisted (semantically) of a single functionality deletion modification, we implemented a new system, the *Kali* automatic patch generation system, that focuses only on removing functionality. Kali generates patches that either 1) delete a single line or block of code, 2) replace an if condition with true or false (forcing the then or else branch to always execute), or 3) insert a single return statement into a function body. Kali accepts a patch if it generates correct outputs on all inputs in the validation test suite. Our hypothesis was that by focusing directly on functionality removal, we would be able to obtain a simpler system that was at least as effective in practice.

RQ5: How effective is Kali in comparison with existing generate-and-validate patch generation systems?

Although we do not advocate using Kali for automatic patch generation, our results show that Kali is more effective than GenProg, RSRepair, and AE. Specifically, Kali finds correct patches for at least as many defects (3 for Kali vs. 3 for AE and 2 for GenProg and RSRepair) and plausible patches for at least as many defects (27 for Kali vs. 18 for GenProg, 10 for RSRepair, and 27 for AE). And Kali works with a simpler and more focused search space. All of our results including the test cases and the semantic analysis of each plausible but incorrect patch are available at [84].

2.2 Overview of Analyzed Systems

GenProg: GenProg combines three basic modifications, specifically delete, insert, and replace, into larger patches, then uses genetic programming to search the resulting patch space. We work with the GenProg system used to perform a "systematic study of automated program repair" that "includes two orders of magnitude more" source code, test cases, and defects than previous studies [55]. As of the time of writing this dissertation, the relevant GenProg paper is referenced on the GenProg web site as the recommended starting point for researchers interested in learning more about GenProg [6]. The GenProg patch evaluation infrastructure works with the following kinds of components [5, 7, 8]:

- **Test Cases:** Individual tests that exercise functionality in the patched application. Examples include php scripts (which are then evaluated by a patched version of php), bash scripts that invoke patched versions of the libtiff tools on specific images, and perl scripts that generate HTTP requests (which are then processed by a patched version of lighttpd).
- **Test Scripts:** Scripts that run the application on a set of test cases and report either success (if the application passes all of the test cases) or failure (if the application does not pass at least one test case).
- **Test Harnesses:** Scripts or programs that evaluate candidate patches by running the relevant test script or scripts on the patched application, then reporting the results (success or failure) back to GenProg.

It is our understanding that the test cases and test scripts were adopted from the existing software development efforts for each of the benchmark GenProg applications and implemented by the developers of these projects for the purpose of testing code written by human developers working on these applications. The test harnesses were implemented by the GenProg developers as part of the GenProg project.

A downloadable virtual machine [8], all of the patches reported in the relevant GenProg paper (these include patches from 10 GenProg executions for each defect) [7], source code for each application, test cases, test scripts, and the GenProg test harness for each application [5] are all publicly available. Together, these components make it possible to apply each patch and run the test scripts or even the patched version of each application on the provided test cases. It is also possible to run GenProg itself.

RSRepair: The goal of the RSRepair project was to compare the effectiveness of genetic programming with random search [82]. To this end, the RSRepair system built on the GenProg system, using the same testing and patch evaluation infrastructure but changing the search algorithm from genetic search to random search. RSRepair was evaluated on 24 of the 55 defects that GenProg was reported to repair [55, 82]. The reported patches are publicly available [12]. For each defect, the RSRepair paper reports patches from 100 runs. We analyze the first 5 patches from these 100 runs.

AE: AE is an extension to GenProg that uses a patch equivalence analysis to avoid repeated testing of patches that are syntactically different but equivalent (according to an approximate patch equivalence test) [104]. AE focuses on patches that only perform one edit and exhaustively enumerates all such patches. The AE experiments were "designed for direct comparison with previous GenProg results" [55, 104] and evaluate AE on the same set of 105 defects. The paper reports one patch per repaired defect, with the patches publicly available [1]. AE is based on GenProg and we were able to leverage the developer test scripts available in the GenProg distribution to compile and execute the reported AE patches.

2.3 Plausibility Analysis

The basic principle behind the GenProg, RSRepair, and AE systems is to generate patches that produce correct results for all of the inputs in the test suite used to validate the patches. We next present our analysis results for the first research question:

RQ1: Do the reported GenProg, RSRepair, and AE patches produce correct results for the inputs in the test suite used to validate the patch?

To investigate this question, we downloaded the reported patches and validation test suites [1, 5, 7, 8, 12]. We then applied the patches, recompiled the patched applications, ran the patched applications on the inputs in the validation test suites, and compared the outputs with the correct outputs. Our results show that the answer to RQ1 is that the majority of the reported GenProg, RSRepair, and AE patches do not produce correct outputs for the inputs in the validation test suite:

- **GenProg:** Of the reported 414 GenProg patches, only 110 are plausible — the remaining 304 generate incorrect results for at least one input in the test suite used to validate the patch. This leaves 18 defects with at least one plausible patch.

- **RSRepair:** Of the analyzed 120 AE patches, only 44 are plausible — the remaining 76 generate incorrect results for at least one input in the test suite used to validate the patch. This leaves 10 defects with at least one plausible patch.
- **AE:** Of the reported 54 AE patches, only 27 are plausible — the remaining 27 generate incorrect results for at least one input in the test suite. This leaves 27 defects with at least one plausible patch.

Test Harness Issues: The GenProg 2012 paper reports that GenProg found successful patches for 28 of 44 defects in php [55]. The results tarball contains a total of 196 patches for these 28 defects. Only 29 of these patches (for 5 of the 44 defects, specifically defects php-bug-307931-307934, php-bug-309892-309910, php-bug-309986-310009, php-bug-310011-310050, and php-bug-310673-310681) are plausible. GenProg accepts the remaining 167 patches because of integration issues between the GenProg test harness and the developer test script.

For php, the developer test script is also written in php. The GenProg test harness executes this developer test script using the version of php with the current GenProg patch under evaluation applied, not the standard version of php. The current patch under evaluation can therefore influence the behavior of the developer test script (and not just the behavior of the test cases).

The GenProg test harness does not check that the php patches cause the developer test scripts to produce the correct result. It instead checks only that the higher order 8 bits of the exit code from the developer test script are 0. This can happen if 1) the test script itself crashes with a segmentation fault (because of an error in the patched version of php that the test case exercises), 2) the current patch under evaluation causes the test script (which is written in php) to exit with exit code 0 even though one of the test cases fails, or 3) all of the test cases pass. Of the 167 accepted patches, 138 are implausible — only 29 pass all of the test cases.

We next present relevant test infrastructure code. The GenProg test harness is written in C. The following lines determine if the test harness accepts a patch. Line

8564 runs the test case and shifts off the lower 8 bits of the exit code. Line 8566 accepts the patch if the remaining upper 8 bits of the exit code are zero.

```
php-run-test.c:
8559 sprintf(buffer,
        "./sapi/cli/php ../php-helper.php -p ./sapi/cli/php -q %s", name);
...
8564 int res = system(buffer) >> 8
8565
8566 if (res == 0) { /* accept patch */
8567     printf("PASS: %s\n", name);
8568     return 0;
8569 } else {
8570     printf("FAIL: %s\n", name);
8571     return 1;
8572 }
```

Note that the `buffer` contains the shell command to run a test, where `name` variable contains the test case name, `./sapi/cli/php` is the patched version of the php interpreter. This patched version is used both to run the php test file for the test case and the `php-helper.php` script that runs the test case.

Test Script Issues: The GenProg libtiff test scripts do not check that the test cases produce the correct output. They instead use a *weak proxy* that checks only that the exercised libtiff tools return exit code 0 (it is our understanding that the libtiff developers, not the GenProg developers, developed these test scripts). The test scripts may therefore accept patches that do not produce the correct output. There is a libtiff test script for each test case; 73 of the 78 libtiff test scripts check only the exit code. This issue causes GenProg to accept 137 implausible libtiff patches (out of a total of 155 libtiff patches). libtiff and php together account for 322 of the total 414 patches that the GenProg paper reports [55].

One of the gmp test scripts does not check that all of the output components are correct (despite this issue, both gmp patches are plausible).

AE: The reported AE patches exhibit plausibility problems that are consistent with the use of weak proxies in the GenProg testing infrastructure. Specifically, only 5 of the 17 reported libtiff patches and 7 of the reported 22 php patches are plausible. Upon further examination of the AE infrastructure, AE eliminated one of the sources of error in the php test harness. Specifically, the AE php test harness no longer shifts off the lower 8 bits of the exit code. ("`>> 8`" is removed in line 8564 of the `php-run-test.c` file). The other php and libtiff weak proxy errors remain in place.

RSRepair: RSRepair uses the same testing infrastructure as GenProg [82]. Presumably because of weak proxy problems inherited from the GenProg testing infrastructure, the reported RSRepair patches exhibit similar plausibility problems. Only 5 of the 75 RSRepair libtiff patches are plausible. All of these 5 patches repair the same defect, specifically libtiff-bug-d13be72c-ccadf48a. The RSRepair paper reports patches for only 1 php defect, specifically php-bug-309892-309910, the 1 php defect for which all three systems are able to generate a correct patch.

ManyBugs: The ManyBugs testing infrastructure exhibits similar errors as the AE test infrastructure [10, 54]. Specifically, the php test harness still uses the patched version of php to run the test cases and the libtiff test scripts check only for exit code 0, not for the correct output. The ManyBugs paper therefore (Figure 4) reports accepted patches that produce incorrect outputs for inputs in the validation test suite. And the ManyBugs php and libtiff testing infrastructure still accepts the following program as a correct implementation of both php and libtiff:

```
int main() { return 0; }
```

Like the AE testing infrastructure (but unlike the earlier GenProg testing infrastructure [55]), the ManyBugs testing infrastructure does not shift off the lower 8 bits of the program exit code. Because of the elimination of this error, the ManyBugs paper reports fewer implausible GenProg patches than the earlier ICSE 2012 paper [55]. Specifically, for the same set of 44 php defects present in both the GenProg 2012 [55] and ManyBugs [54] benchmark sets, the reported number of cases for which GenProg generates plausible patches has dropped from 28 in the ICSE 2012 paper [55] to 20 in the ManyBugs paper [54].

2.4 Correctness Analysis

We analyze each plausible patch in context to determine if it correctly repairs the defect. We next present our analysis results for the second research question:

RQ2: Are any of the reported GenProg, RSRepair, and AE patches correct?

Patch Correctness Results: Our analysis indicates that only 5 of the 414 GenProg patches (3 for python-bug-69783-69784 and 2 for php-bug-309892-309910) are correct. This leaves GenProg with correct patches for 2 out of 105 defects. Only 4 of the 120 RSRepair patches (2 for python-bug-69783-69784 and 2 for php-bug-309892-309910) are correct. This leaves RSRepair with correct patches for 2 out of 24 defects. Only 3 of the 54 AE patches (1 for php-bug-309111-309159, 1 for php-bug-309892-309910, and 1 for python-bug-69783-69784) are correct. This leaves AE with correct patches for 3 out of 54 defects.

For each plausible but incorrect patch that GenProg or AE generate, and each plausible but incorrect RSRepair patch that we analyze, we developed a new test case that exposes the defect in the incorrect patch [84].

Patch Correctness Clarity: We acknowledge that, in general, determining whether a specific patch corrects a specific defect can be difficult (or in some cases not even well defined). We emphasize that this is not the case for the patches and defects that we consider here. The correct behavior for all of the defects is clear, as is patch correctness and incorrectness.

Developer Patch Comparison: For each defect, the GenProg benchmark suite identifies a corrected version of the application that does not have the defect. In most cases the corrected version is a later version produced by a developer writing an explicit developer patch to repair the error. In other cases the corrected version simply applies a deliberate functionality change — there was no defect in the original

```
 1  -if (y < 1000) {
 2  -  tmp___0 = PyDict_GetItemString(moddict,
 3  -              "accept2dyear");
 4  -  accept = tmp___0;
 5  -  if ((unsigned int)accept !=
 6  -      (unsigned int )((void *)0)) {
 7  -    tmp___1 = PyObject_IsTrue(accept);
 8  -    acceptval = tmp___1;
 9  -    if (acceptval == -1) {
10  -      return (0);
11  -    } else { }
12  -    if (acceptval) {
13  -      if (0 <= y) {
14  -        if (y < 69) {
15  -          y += 2000;
16  -        } else {
17  -          goto _L;
18  -        }
19  -      } else {
20  -        _L: /* CIL Label */
21  -        if (69 <= y) {
22  -          if (y < 100) {
23  -            y += 1900;
24  -          } else {
25  -            PyErr_SetString(
26  -              PyExc_ValueError,
27  -              "year out of range");
28  -            return (0);
29  -          }
30  -        } else {
31  -          PyErr_SetString(
32  -            PyExc_ValueError,
33  -            "year out of range");
34  -          return (0);
35  -        }
36  -      }
37  -      tmp___2 =
38  -       PyErr_WarnEx(
39  -         PyExc_DeprecationWarning,
40  -"Century info guessed for a 2-digit year.",
41  -         1);
42  -      if (tmp___2 != 0) {
43  -        return (0);
44  -      } else { }
45  -    } else { }
46  -  } else {
47  -    return (0);
48  -  }
49  -} else { }
50  p->tm_year = y - 1900;
51  (p->tm_mon) --;
52  p->tm_wday = (p->tm_wday + 1) % 7;
```

Figure 2-1: GenProg Patch for python-bug-69783-69784. Delete Lines 1-49.

```
1  -if (y < 1000) {
2  -  PyObject *accept = PyDict_GetItemString(
3  -    moddict, "accept2dyear");
4  -  if (accept != NULL) {
5  -    int acceptval=PyObject_IsTrue(accept);
6  -    if (acceptval == -1)
7  -      return 0;
8  -    if (acceptval) {
9  -      if (0 <= y && y < 69)
10 -        y += 2000;
11 -      else if (69 <= y && y < 100)
12 -        y += 1900;
13 -      else {
14 -        PyErr_SetString(PyExc_ValueError,
15 -          "year out of range");
16 -        return 0;
17 -      }
18 -      if (PyErr_WarnEx(
19 -            PyExc_DeprecationWarning,
20 -"Century info guessed for a 2-digit year.",
21 -            1) != 0)
22 -        return 0;
23 -      }
24 -  }
25 -  else
26 -    return 0;
27 -}
28  p->tm_year = y - 1900;
29  p->tm_mon--;
30  p->tm_wday = (p->tm_wday + 1) % 7;
```

Figure 2-2: Developer Patch for python-bug-69783-69784. Delete Lines 1-27.

version of the application. In yet other cases the identified correct version is an earlier version of the application. In these cases, it is possible to derive an implicit developer patch that reverts the application back to the earlier version.

Our analysis indicates that the developer patches are, in general, consistent with our correctness analysis. Specifically, 1) if our analysis indicates that the reported GenProg, RSRepair, or AE patch is correct, then the patch has the same semantics as the developer patch, 2) if our analysis indicates that the reported GenProg, RSRepair, or AE patch is not correct, then the patch has different semantics than the developer patch, and 3) if we developed a new test case to invalidate generated plausible but incorrect patches for a defect, the corresponding developer patched version of the application produces correct output for the new input.

python-bug-69783-69784: Note that python-bug-69783-69784 is in fact not a bug. It instead corresponds to a deliberate functionality change. The removed code implemented python support for two-year dates. This functionality was deliberately removed by the developer in revision 69784. Because the original code correctly implemented the two-digit date functionality, there is no defect to patch — this bug corresponds to a deliberate functionality change instead of a defect. Figure 2-1 presents the GenProg patch for python-bug-69783-69784. Figure 2-2 presents the developer patch. Both of the patches remove an if statement (lines 1-25 in Figure 2-2, lines 1-52 in Figure 2-1). Because GenProg generates preprocessed code, the GenProg patch is larger than but semantically equivalent to the developer patch. AE and RSRepair also generate correct patches that are semantically equivalent to this GenProg patch.

php-bug-309892-309910: Figure 2-3 presents the GenProg patch for php-bug-309892-309910. Figure 2-4 presents the developer patch. Both of the patches remove an obsolete check implemented by the deleted if statement (lines 14-18 in Figure 2-3 and lines 7-9 in Figure 2-4). AE and RSRepair generate semantically equivalent patches.

php-bug-309111-309159: Figure 2-5 presents the AE patch for php-bug-309111-309159. Figure 2-6 presents the developer patch. php-309111-309159 is an url parsing defect — the PHP function parse_url() may incorrectly parse urls that contain

```
1   if (offset >= (long )s1_len) {
2     php_error_docref0(
3       (char const*)((void *)0),
4       1 << 1L,
5       "The start position cannot exceed "+
6       "initial string length");
7     while (1) {
8       __z___1 = return_value;
9       __z___1->value.lval = 0L;
10      __z___1->type = (unsigned char)3;
11      break;
12    }
13    return;
14  } else { }
15 -if (len > (long )s1_len - offset) {
16 -  len = (long )s1_len - offset;
17 -} else { }
18  if (len) {
19    tmp___1 = len;
20  } else {
21    if ((long )s2_len >
22        (long)s1_len - offset) {
23      tmp___0 = (long )s2_len;
24    } else {
25      tmp___0 = (long )s1_len - offset;
26    }
27    tmp___1 = tmp___0;
28  }
29  cmp_len = (unsigned int )tmp___1;
```

Figure 2-3: GenProg Patch for php-bug-309892-309910. Delete Lines 15-17.

```
1   if (offset >= s1_len) {
2     php_error_docref(NULL TSRMLS_CC,
3       E_WARNING,
4       "The start position cannot " +
5       "exceed initial string length");
6     RETURN_FALSE;
7   }
8  -if (len > s1_len - offset) {
9  -  len = s1_len - offset;
10 -}
11  cmp_len = (uint) (len ? len : MAX(
12     s2_len, (s1_len - offset)));
```

Figure 2-4: Developer Patch for php-bug-309892-309910. Delete Lines 8-10.

```
 1     if (pp) {
 2       if ((unsigned int)pp < (unsigned int)p) {
 3  +      ...
 4         p = pp;
 5  +      ...
 6  +      if (__genprog_mutant == 25) {
 7  +        if (p - s) {
 8  +          tmp___24 = _estrndup(s,
 9  +            (unsigned int )(p - s));
10  +          ret->path = (char *)tmp___24;
11  +          php_replace_controlchars_ex(
12  +            ret->path, p - s);
13  +        } else { }
14  +      }
15  +      ...
16         goto label_parse;
17       } else { }
18     } else { }
19     if (p - s) {
20       tmp___21 = _estrndup(s,
21         (unsigned int )(p - s));
22       ret->path = (char *)tmp___21;
23       php_replace_controlchars_ex(ret->path,
24         p - s);
25     } else { }
```

Figure 2-5: AE Patch for php-bug-309111-309159. Insert lines 5-15.

```
1      if (pp && pp < p) {
2    +   if (pp - s) {
3    +     ret->path = estrndup(s, (pp-s));
4    +     php_replace_controlchars_ex(ret->path,
5    +       (pp - s));
6    +   }
7        p = pp;
8        goto label_parse;
9      }
10     if (p - s) {
11       ret->path = estrndup(s, (p-s));
12       php_replace_controlchars_ex(ret->path,
13         (p - s));
14     }
```

Figure 2-6: Developer Patch for php-bug-309111-309159. Insert lines 2-6.

question marks. The AE patch (with __genprog_mutant equal to 25) copies the if statement (lines 23-29 of Figure 2-5) to the location after the assignment p = pp. Therefore p is equal to pp when the copied block executes. In this context, the copied block is semantically equivalent to the block that the developer patch adds before the assignment statement. In the AE patch, the code involving __genprog_mutant works with the AE test infrastructure to compile multiple generated patches into the same file for later dynamic selection by the AE test infrastructure.

2.5 GenProg Reexecutions

We next present our analysis results for the third research question:

RQ3: Do stronger test suites enable GenProg to produce more correct patches?

To determine whether GenProg [55] is able to generate correct patches if we correct the issues in the patch evaluation infrastructure and provide GenProg with stronger test suites, we perform the following GenProg reexecutions:

Corrected Patch Evaluation Infrastructure: We first corrected the GenProg patch evaluation infrastructure issues (see Section 2.3). Specifically, we modified the php test harness to ensure that the harness correctly runs the test script and correctly reports the results back to GenProg. We strengthened the 73 libtiff test scripts to, as appropriate, compare various metadata components and/or the generated image output with the correct output. We modified the gmp test scripts to check all output components.

Augmented Test Suites: We augmented the GenProg test suites to include the new test cases (see Section 2.4) that expose the defects in the plausible but incorrect GenProg patches.

GenProg Reexecution: For each combination of defect and random seed for which GenProg generated an incorrect patch, we reexecuted GenProg with that same combination. These reexecutions used the corrected patch evaluation infrastructure and the augmented test suites.

Results: These reexecutions produced 13 new patches (for defects libtiff-bug-5b02179-3dfb33b and lighttpd-bug-2661-2662). Our analysis indicates that the new patches that GenProg generated for these two defects are plausible but incorrect. We therefore developed two new test cases that exposed the defects in these new incorrect patches. We included these new test cases in the test suites and reexecuted GenProg again. With these test suites, the GenProg reexecutions produced no patches at all. The new test cases are available [84].

2.6 Semantic Patch Analysis

For each plausible patch, we manually analyzed the patch in context to determine if it is semantically equivalent to either 1) the deletion of a single statement or block of code, or 2) the insertion of a single return or exit statement. This analysis enables us to answer the fourth research question:

RQ4: Are the reported GenProg, RSRepair, and AE patches equivalent to a single modification that simply deletes functionality?

Our analysis indicates that the overwhelming majority of the reported plausible patches are equivalent to a single functionality deletion modification. Specifically, 104 of the 110 plausible GenProg patches, 37 of the plausible 44 RSRepair patches, and 22 of the plausible 27 AE patches are equivalent to a single deletion or return insertion modification. Note that even though AE contains analyses that attempt to determine if two patches are equivalent, the analyses are based on relatively shallow criteria (syntactic equality, dead code elimination, and equivalent sequences of independent instructions) that do not necessarily recognize the functionality deletion equivalence of syntactically complex sequences of instructions. Indeed, the AE paper, despite its focus on semantic patch equivalence, provides no indication that the overwhelming majority of the reported patches are semantically equivalent to a single functionality deletion modification [104].

2.6.1 Weak Test Suites

During our analysis, we obtained a deeper understanding of why so many plausible patches simply delete functionality. A common scenario is that one of the test cases exercises a defect in functionality that is otherwise unexercised. The patch simply deletes functionality that the test case exercises. This deletion then impairs or even completely removes the functionality.

These results highlight the fact that *weak test suites* — i.e., test suites that provide relatively limited coverage — may be appropriate for human developers (who operate with a broader understanding of the application and are motivated to produce correct patches) but (in the absence of additional techniques designed to enhance their ability to produce correct patches) not for automatic patch generation systems that aspire to produce correct patches.

2.6.2 Case Studies

We next present several case studies of patches for specific bugs.

libtiff-bug-0860361d-1ba75257: Revision number 0860361d of libtiff incorrectly implements an integer overflow check at TIFFFetchData() in tif_dirread.c. Figure 2-7 presents the source code snippet and the developer patch for this defect. Figure 2-8 presents the GenProg and AE patches for this error. The check at line 9 in Figure 2-7 is incorrect and classifies many benign tiff images as bad images. The developer patch

```
 1   static tsize_t
 2   TIFFFetchData(TIFF* tif,
 3     TIFFDirEntry* dir, char* cp) {
 4     int w = TIFFDataWidth(
 5      (TIFFDataType) dir->tdir_type);
 6     tsize_t cc = dir->tdir_count * w;
 7     /* Check for overflow. */
 8     if (!dir->tdir_count || !w ||
 9 -     (tsize_t)dir->tdir_count / w != cc)
10 +     cc / w != (tsize_t)dir->tdir_count)
11       goto bad;
12     ...
13   }
```

Figure 2-7: Developer Patch for libtiff-bug-0860361d-1ba75257. Modify Lines 9-10.

```
 1   static tsize_t
 2   TIFFFetchData(TIFF* tif,
 3     TIFFDirEntry* dir, char* cp) {
 4     int w = TIFFDataWidth(
 5      (TIFFDataType) dir->tdir_type);
 6     tsize_t cc = dir->tdir_count * w;
 7     /* Check for overflow. */
 8 -   if (!dir->tdir_count || !w ||
 9 -    (tsize_t)dir->tdir_count / w != cc)
10 -     goto bad;
11     ...
12   }
```

Figure 2-8: GenProg and AE Patches for libtiff-bug-0860361d-1ba75257. Delete Lines 8-10.

```
 1   static int statbuf_from_array(zval *array,
 2     php_stream_statbuf *ssb) {
 3     zval **elem;
 4  #define ENTRY_EX(name, name2)                     \
 5  if (SUCCESS == zend_hash_find(array,              \
 6    #name, sizeof(#name), (void**)&elem)) {         \
 7  +  SEPARATE_ZVAL(elem);                           \
 8     convert_to_long(*elem);                        \
 9    ssb->sb.st_##name2 = Z_LVAL_PP(elem); }
10  #define ENTRY(name) ENTRY_EX(name,name)
11
12     ENTRY(dev);
13     ENTRY(...);
14     ...
15   }
```

Figure 2-9: Developer Patch for php-bug-307931-307934. Insert Line 7.

fixes the overflow check (see lines 9-10 in Figure 2-7). All plausible GenProg and AE patches remove either the integer overflow check or even the entire branch condition completely (see lines 8-10 in Figure 2-8), i.e., the patches effectively reintroduce the integer overflow vulnerability (CVE-2006-2025). The patches pass all of the test cases in the validation test suite because the test suite does not contain an overflow triggering image. We obtained an existing proof-of-concept tiff image for CVE-2006-2025. This image exposes the vulnerability that the GenProg and AE patches reintroduce. The reported GenProg and AE patches in the ManyBugs paper suffer from the same problem, reintroduce the same vulnerability, and are exposed by the same image reported in CVE-2006-2025 [10].

php-bug-307931-307934: The PHP interpreter 5.3.5 has a defect (Bug #53903) in its implementation of the fstat() function (userspace.c:852). PHP allows PHP programs to define streamWrapper classes to customize the behavior of many I/O stream functions including fstat(). When a PHP program registers a streamWrapper class to an opened stream, fstat() should invoke the corresponding function in the streamWrapper class. However, the implementation of fstat() contains an error when it collects the results returned from the invoked streamWrapper function. Specifically, it does not correctly separate the underlying data structures for the result values during a conversion operation, so that multiple PHP values incorrectly share a single data

```
1      static int php_userstreamop_stat(
2        php_stream *stream,
3        php_stream_statbuf *ssb) {
4        ...
5        call_result = call_user_function_ex(NULL,
6          &us->object, &func_name,
7          &retval, 0, NULL, 0, NULL);
8        if (call_result == SUCCESS &&
9          retval != NULL &&
10         Z_TYPE_P(retval) == IS_ARRAY) {
11  -      if (0 == statbuf_from_array(
12  -        retval, ssb))
13           ret = 0;
14       } else { ... }
15       ...
16     }
```

Figure 2-10: GenProg and AE Patches for php-bug-307931-307934. Delete Lines 11-12.

structure in the memory. This incorrect sharing eventually corrupts the PHP internal data structures. The result is that PHP fails to correctly execute the remaining PHP statements following the fstat() invocation.

Figure 2-9 presents the developer patch for this defect. The developer changes the function statbuf_from_array(), which converts the results in the array buffer (returned by the streamWrapper routine), to an ssb struct. At line 7, the developer modifies the macro so that whenever it copies a PHP value into an ssb struct field, it correctly separates the copied value.

Figure 2-10 presents example GenProg and AE patches for this defect. Instead of fixing the defect in the conversion operation, all plausible GenProg and AE patches perform functionality elimination. The patches either completely delete the dispacher code that invokes the streamWrapper routine or delete the invocation of the error triggering conversion operation (see lines 11-12 in Figure 2-10). The patches effectively turn fstat() to a no-op on a stream with a registered streamWrapper class. Such functionality elimination patches pass all of the test cases in the valdation PHP test suite because the test suite does not contain a test case that checks the normal functionality of fstat() with streamWrapper classes. We crafted a test case that checks this normal behavior. On this test case, all plausible GenProg and AE patches produce different outputs than the later developer patch. We also ran the crafted test case on the GenProg and AE patches reported in the ManyBugs paper. The ManyBugs patches also produce different outputs than the developer patch.

php-bug-310673-310681: The PHP interpreter version 5.3.6 has a defect (BUG #54623) in its I/O stream resource management code. At php_stream_from_persistent_id() in streams.c:117, it uses a global linked list (i.e. regular_list) to store all allocated resources. But the implementation fails to maintain the invariant that each resource can appear in the list only once. Multiple occurences of the same resource in the list cause a segmentation fault when the PHP interpreter deallocates resources from this list.

Figure 2-11 presents the developer patch for this defect. The patch uses a while loop (see lines 16-27 in Figure 2-11) to check whether the resource has already been loaded into the global linked list. If so, the patch does not add the resource into the list (see lines 35-38 in Figure 2-11).

Figure 2-12 presents example GenProg and AE patches. Instead of fixing the code to maintain the invariant, all plausible GenProg and AE patches delete the assignment statement that stores allocated resource ids into the list (see line 11-13 in Figure 2-12). The patches introduce memory leaks because they effectively turn the memory deallocation operations that operate on the list into no-ops. The patches pass all of the test cases in the validation test suite because no test case checks the memory deallocation component. We crafted a test case that allocates and deallocates four million I/O sockets. In our environment, this test case causes all plausible GenProg and AE patches run out of memory. The later developer patch executes this test case correctly in the same environment. We also ran the crafted test case on the AE patch reported in the ManyBugs paper (the ManyBugs paper reports no plausible GenProg patch for this defect). The test case still causes the patched PHP to run out of memory.

gzip-bug-3fe0ca-39a362: gzip version 1.3 fails to terminate when extracting a carefully crafted, malformed archive (huft-segv.tgz) that uses a dynamic Huffman encoding scheme. The program hangs because not enough invalid codes are allocated

```
 1       int php_stream_from_persistent_id(
 2         const char *persistent_id,
 3         php_stream **stream) {
 4         ...
 5         if (zend_hash_find(&EG(persistent_list),
 6           (char*)persistent_id, ...) == SUCCESS){
 7           ...
 8           if (stream) {
 9   +         ...
10   +         ulong index = -1;
11   +         /* see if this persistent resource
12   +          * already has been loaded to the
13   +          * regular list;*/
14   +         zend_hash_internal_pointer_reset_ex(
15   +           &regular_list, &pos);
16   +         while (zend_hash_get_current_data_ex(
17   +           &regular_list,
18   +           &regentry, &pos) == 0) {
19   +           if (regentry->ptr == le->ptr) {
20   +             zend_hash_get_current_key_ex(
21   +               &regular_list, NULL, NULL,
22   +               &index, 0, &pos);
23   +             break;
24   +           }
25   +           zend_hash_move_forward_ex(
26   +             &regular_list,&pos);
27   +         }
28             *stream = (php_stream*)le->ptr;
29   +         if (index == -1) {
30   +           /* not found in regular_list */
31               le->refcount++;
32               (*stream)->rsrc_id =
33                ZEND_REGISTER_RESOURCE(NULL,
34                  *stream, le_pstream);
35   +         } else {
36   +           regentry->refcount++;
37   +           (*stream)->rsrc_id = index;
38   +         }
39           }
40           return PHP_STREAM_PERSISTENT_SUCCESS;
41           ... }
42         return PHP_STREAM_PERSISTENT_NOT_EXIST;
43       }
```

Figure 2-11: Developer Patch for php-bug-310673-310681. Insert Lines 9-30 and Lines 35-38.

```
 1     int php_stream_from_persistent_id(
 2       const char *persistent_id,
 3       php_stream **stream) {
 4       ...
 5       if (zend_hash_find(&EG(persistent_list),
 6         (char*)persistent_id, ...) == SUCCESS) {
 7         ...
 8         if (stream) {
 9           *stream = (php_stream*)le->ptr;
10           le->refcount++;
11   -       (*stream)->rsrc_id =
12   -       ZEND_REGISTER_RESOURCE(NULL,
13   -         *stream, le_pstream);
14         }
15         return PHP_STREAM_PERSISTENT_SUCCESS;
16         ... }
17       return PHP_STREAM_PERSISTENT_NOT_EXIST;
18     }
```

Figure 2-12: GenProg and AE Patches for php-bug-310673-310681. Delete Lines 11-13.

```
 1   int huft_build(b, n, s, d, e, t, m){
 2     ...
 3     memset((voidp)(c),0,(sizeof(c)));
 4     p=b; i=n;
 5     do {
 6       c[*p]++; p++;
 7     } while (--i);
 8     if (c[0] == n)
 9     {
10   -   q=(struct huft *)malloc(2*sizeof(*q));
11   +   q=(struct huft *)malloc(3*sizeof(*q));
12       if (!q)
13         return 3;
14   -   hufts+=2
15   +   hufts+=3;
16       q[0].v.t=(struct huft *)((void*)0);
17       q[1].e=99; q[1].b=1;
18       /*invalid code number*/
19   +   q[2].e=99; q[2].b=1;
20       *t=q + 1; *m=1;
21       return 0;
22     }
23     ...
24   }
```

Figure 2-13: Developer Patch for gzip-bug-3fe0ca-39a362. Modify Lines 10-11, Lines 14-15, and Line 19.

```
1  int huft_build(b, n, s, d, e, t, m) {
2    ...
3  + int tmp___8;
4    memset((voidp)(c),0,(sizeof(c)));
5    p=b; i=n;
6    do {
7      c[*p]++; p++;
8    } while (--i);
9    if (c[0] == n)
10   {
11     q=(struct huft*)malloc(2*sizeof(*q));
12     if (!q)
13       return 3;
14 +   else
15 +     return tmp__8;
16     hufts+=2
17     q[0].v.t=(struct huft *)((void*)0);
18     q[1].e=99; q[1].b=1;
19     *t=q + 1; *m=1;
20     return 0;
21   }
22   ...
23 }
```

Figure 2-14: GenProg Patch for gzip-bug-3fe0ca-39a362. Insert Line 3 and Lines 14-15.

in the Huffman table data structure, triggering an infinite loop in the inflate_codes() routine in inflate.c.

Figure 2-13 presents the developer patch for this defect. The developer patch for this bug allocates an additional invalid code to the Huffman table, allowing the program to exit gracefully with an 'invalid compressed data -format violated'

Figure 2-14 presents a representative AE or genprog patch for this defect. The AE patch and all but one of the genprog patches circumvents the infinite loop only when the MALLOC_PERTURB environment variable is set. When MALLOC_PERTURB is set, all allocated memory is initialized to the non-zero value contained in MALLOC_PERTURB (in the gzip test suite, this value is set to 67). The goal is to break any behavior that depends on uninitialized memory being set to zero. In production, MALLOC_PERTURB is unset and newly allocated memory is initialized to zero in glibc. When run in production, all but one of the genprog patches enter an infinite loop (specifically on the huft-segv.tgz test case) — they rely on MALLOC_PERTURB being set to a nonzero value to generate correct results for the test cases in the test suite. The one remaining genprog patch does not enter an infinite loop, produces correct outputs for the inputs in the test suite, but fails to implement basic gzip functionality such as extracting normal archives containing more than zero characters. When used to extract an archive containing the text "AB," for example, the patched version of gzip produces non-printable characters. This patch is accepted because of a weak test gzip test suite that does not contain test cases that test common case functionality.

gzip-bug-a1d3d4-f17cbd: gzip version 1.8 fails to interpret the '-' argument as stdin if it is not the first argument on the command line.

Figure 2-15 presents the developer patch for this defect. The developer patch for this defect initializes the ifd variable to the file descriptor pointing to stdin, ensuring the program doesn't terminate early if it's reading from standard input.

Figure 2-16 presents the AE patch for this defect. The AE patch initializes the ifd variable to the istat file descriptor, which points to one of the input files. If the archive

```
 1  static void treat_stdin(){
 2       ...
 3       clear_bufs();
 4       to_stdout = 1;
 5       part_nb = 0;
 6  +    ifd = fileno(stdin);
 7       if (decompress) {
 8          method = get_method(ifd);
 9          if (method < 0) {
10            do_exit(exit_code);
11          }
12       }
13       if (list) {
14            do_list(ifd, method);
15            return;
16       }
17       ...
18  }
```

Figure 2-15: Developer Patch for gzip-bug-a1d3d4-f17cbd. Insert Line 6.

```
 1  static void treat_stdin(){
 2       ...
 3       clear_bufs();
 4       to_stdout = 1;
 5       part_nb = 0;
 6       if (decompress) {
 7          method = get_method(ifd);
 8          if (method < 0) {
 9             do_exit(exit_code);
10          }
11       }
12       if (list) {
13            do_list(ifd, method);
14            return;
15       }
16       ...
17  +    ifd = open_input_file(iname, & istat);
18  }
```

Figure 2-16: AE Patch for gzip-bug-a1d3d4-f17cbd. Insert Line 17.

supplied via stdin is different from the input archive, the repaired gzip program uses the input archive for both sources. For example, given the following set of commands:

```
echo "hello" | gzip > input1.gz
echo "world" | gzip > input2.gz
gzip -dc input1.gz - < input2.gz
```

The expected output is helloworld but the output produced by the repaired program is hellohello. The gzip test suite contains a test case that is designed to detect an incorrect implementation of the '-' argument. The generated patch passes this test case because, in this gzip test case, the first and second input files are the same.

2.6.3 Impact of Functionality Deletion Patches

Our analysis of the patches also indicated that (in contrast to previously reported results [56]) the combination of test suites with limited coverage and support for functionality deletion can promote the generation of patches with negative effects such as the introduction of security vulnerabilities and the elimination of standard functionality.

Check Elimination: Several defects are caused by incorrectly coded checks. The test suite contains a test case that causes the check to fire incorrectly, but there is no test case that relies on the check to fire correctly. The generated patches simply remove the check. The consequences vary depending on the nature of the check. For example:

- **Integer Overflow:** libtiff-bug-0860361d-1ba75257 incorrectly implements an integer overflow check. The generated patches remove the check, in effect reintroducing a security vulnerability from a previous version of libtiff (CVE-2006-2025) that a remote attacker can exploit to execute arbitrary injected code [3].
- **Buffer Overflow:** Defect fbc-bug-5458-5459 corresponds to an overly conservative check that prevents a buffer overflow. The generated patches remove the check, enabling the buffer overflow.

Standard Feature Elimination: Defects php-bug-307931-307934, gzip-bug-3fe0ca-39a362, lighttpd-bug-1913-1914, lighttpd-bug-2330-2331 correspond to incorrectly handled cases in standard functionality. The test suite contains a test case that exposes the incorrectly handled case, but no test case that exercises the standard functionality. The patches impair or remove the functionality, leaving the program unable to process standard use cases (such as decompressing non-zero files or initializing associative array elements to integer values).

Undefined Accesses: Patches often remove initialization code. While the resulting undefined accesses may happen to return values that enable the patch to pass the test cases, the patches can be fragile — different environments can produce values that cause the patch to fail (e.g., the AE patch for fbc-bug-5458-5459).

Deallocation Elimination: The patches for wireshark-bug-37112-37111, php-bug-310673-310681, and php-bug-310011-310050 eliminate memory management errors by removing relevant memory deallocations. While this typically introduces a memory leak, it can also enhance survival by postponing the failure until the program runs out of memory (which may never happen). We note that human developers often work around difficult memory management defects by similarly removing deallocations.

Survival Enhancement: One potential benefit of even incorrect patches is that they may enhance the survival of the application even if they do not produce completely correct execution. This was the goal of several previous systems (which often produce correct execution even though that was not the goal) [25, 28, 35, 50, 61, 71, 74, 80, 90]. Defect lighttpd-bug-1794-1795 terminates the program if it encounters an unknown configuration file setting. The generated patches enhance survival by removing the check and enabling lighttpd to boot even with such configuration files. We note that removing the check is similar to the standard practice of disabling assertions in production use.

Relatively Minor Defects: We note that some of the defects can be viewed as relatively minor. For example, python-bug-69223-69224 causes the unpatched version of python to produce a SelectError message instead of a ValueError message — i.e., the correct behavior is to produce an error message, the defect is that python produces the wrong error message. Three of the wireshark defects (wireshark-bug-37172-37171, wireshark-bug-37172-37173, wireshark-bug-37284-37285) were caused by a developer checking in a version of wireshark with a debug macro flag set. The relevant defect is that these versions generate debugging information to the screen and to a log file. The correct behavior omits this debugging information.

2.7 Original GenProg Patches

We also analyzed the reported patches from the original GenProg system [40, 103]. Out of the 11 defects evaluated in the two papers, the corresponding patches for 9 defects are plausible but incorrect. 9 of the 11 patches simply eliminate functionality (by removing statements or adding a return or exit statements). We next discuss the reported patch for each application in turn.

- **uniq:** The patch is semantically equivalent to removing the statement *buf++ = c at uniq.c:74. The effect is that the patched application will ignore the user input file and operate as if the file were empty. Because of the use of a weak proxy, this patch is not plausible. The test scripts check the exit code of the program, not whether the output is correct.

- **look-u:** The patch is semantically equivalent to removing the condition argv[1] == "-" from the while loop at look.c:63. The effect is that look will treat the first command line argument (-d, -f, -t) as the name of the input file. Unless a file with such a name exists, look will then immediately exit without processing the intended input file.

- **look-s:** The patch is semantically equivalent to replacing the statement mid = (top+bot)/2 at look.c:87 with exit(0). The effect is that look will always exit immediately without printing any output (if the input file exists, if not, look will print an error message before it exits).

  The patch is plausible because the use of a weak test suite — the correct output for the positive and negative test cases is always no output. If the correct output for any of the test cases had been non-empty, this patch would have been implausible.

- **units:** The units program asks the user to input a sequence of pairs of units (for example, the pair meter and feet) and prints out the conversion factor between the two units in the pair. The patch is semantically equivalent to adding init() after units.c:279. The unpatched version of the program does not check for an overflow of the user input buffer. A long user input will therefore overflow the buffer.

  The GenProg patch does not eliminate the buffer overflow. It instead clears the unit table whenever the program reads an unrecognized unit (whether the unit overflows the user input buffer or not). Any subsequent attempt to look up the conversion for any pair of units will fail. It is also possible for the buffer overflow to crash the patched program.

- **deroff:** When deroff reads a backslash construct (for example, \L), it should read the next character (for example, ") as a delimiter. It should then skip any text until it reaches another occurrence of the delimiter or the end of the line.

  The patch is semantically equivalent to removing the statement bdelim=c (automatically generated as part of a macro expansion) at deroff.c:524. The effect is that the patched program does not process the delimiter correctly — when it encounters a delimiter, it skips all of the remaining text on the line, including any text after the next occurrence of the delimiter.

- **nullhttpd:** The patch is semantically equivalent to removing the call to strcmp(..., "POST") at httpd_comb.c:4092-4099. The effect is that all POST requests generate an HTML bad request error reply.

- **indent:** The patch is semantically equivalent to adding a return after indent.c:926. The GenProg 2009 paper states that "Our repair removes handling of C comments that are not C++ comments." Our experiments with the patched version indicate that the patched version correctly handles at least some C comments that are not C++ comments (we never observed a C comment that it handled incorrectly). In many cases, however, the patched version simply exits after reading a { character, truncating the input file after the {.

- **flex:** The patch is semantically equivalent to removing the call to strcpy() at flex_comb.c:13784. This call transfers the token (stored in yytext) into the variable nmdef. Removing the call to strcpy() causes flex to incorrectly operate with an uninitialied nmdef. This variable holds one of the parsed tokens to process. The effect is that flex fails to parse the input file and incorrectly produces error messages that indicate that flex encountered an unrecognized rule.

- **atris:** The commments in the atris source code indicate that it is graphical tetris game. atris has the ability to load in user options stored in the .atrisrc in the user's home directory. A call to sprintf() at atrix_comb.c:5879 initializes the string buffer that specifies the filename of this .atrisrc file. If the home directory is longer than 2048 characters this sprintf() call will overflow the buffer.

  The patch is semantically equivalent to removing the call to sprintf() at atrix_comb.c:5879. The result is that the program passes an uninitialized filename to the procedure that reads the .atrisrc file.

The remaining 2 programs, for which GenProg generates correct patches, are used as motivating examples in the ICSE 2009 and GECCO 2009 papers [40, 103]. These two programs contain less than 30 lines of code.

We note that many of the test scripts use weak proxies. Specifically, all uniq, look-u, and look-s test cases do not compare the output of the patched program to the correct output. They instead check only that the patched program produces the correct exit code. Similarly, the deroff and indent negative test case test scripts only check the exit code.

Defect	GenProg	RSRepair	AE	Kali			
				Result	Search Space	Search Time	Type
fbc-5458-5459	Plausible	-	Plausible‡	Plausible	737	2.4m	SL†
gmp-14166-14167	Plausible	Plausible‡	Plausible	Plausible	1169	19.5m	DP
gzip-3fe0ca-39a362	Plausible	Plausible	Plausible‡	Plausible	1241	28.2m	SF (119)*
gzip-a1d3d4-f17cbd	No Patch	-	Plausible	No Patch			
libtiff-0860361d-1ba75257	Plausible‡	Implausible	Plausible‡	Plausible	1525	16.7m	SL*
libtiff-5b02179-3dfb33b	Plausible‡	Implausible	Plausible‡	Plausible	1476	4.1m	DP
libtiff-90d136e4-4c66680f	Implausible	Implausible	Plausible‡	Plausible	1591	45.0m	SL†
libtiff-d13be72c-ccadf48a	Plausible‡	Plausible‡	Plausible‡	Plausible	1699	42.9m	SL*
libtiff-ee2ce5b7-b5691a5a	Implausible	Implausible	Plausible‡	Plausible	1590	45.1m	SF(10)*
lighttpd-1794-1795	Plausible‡	-	Plausible‡	Plausible	1569	5.9m	
lighttpd-1806-1807	Plausible‡	Plausible‡	Plausible‡	Plausible	1530	55.5m	SF(21)†
lighttpd-1913-1914	Plausible‡	Plausible‡	No Patch	Plausible	1579	158.7m	SL*
lighttpd-2330-2331	Plausible‡	Plausible‡	Plausible‡	Plausible	1640	36.8m	SF(19)†
lighttpd-2661-2662	Plausible‡	Plausible‡	Plausible‡	Plausible	1692	59.7m	DP
php-307931-307934	Plausible‡	-	Plausible‡	Plausible	880	9.2m	DP
php-308525-308529	No Patch	-	Plausible‡	Plausible	1152	234.0m	SL†
php-309111-309159	No Patch	-	Correct	No Patch			
php-309892-309910	Correct‡	Correct‡	Correct‡	Correct	1498	20.2m	C
php-309986-310009	Plausible‡	-	Plausible‡	Plausible	1125	10.4m	SF(27)*
php-310011-310050	Plausible	-	Plausible‡	Plausible	971	12.9m	SL*
php-310370-310389	No Patch	-	No Patch	Plausible	1096	12.0m	DP
php-310673-310681	Plausible‡	-	Plausible‡	Plausible	1295	89.00m	SL*
php-311346-311348	No Patch	-	No Patch	Correct	941	14.7m	C
python-69223-69224	No Patch	-	Plausible	No Patch			
python-69783-69784	Correct‡	Correct‡	Correct‡	Correct	1435	16.1m	C
python-70098-70101	No Patch	-	Plausible	Plausible	1233	6.8m	SL*
wireshark-37112-37111	Plausible‡	Plausible	Plausible‡	Plausible	1412	19.6m	SL†
wireshark-37172-37171	No Patch	-	Plausible‡	Plausible	1459	10.9m	SL†
wireshark-37172-37173	No Patch	-	Plausible‡	Plausible	1459	10.9m	SL†
wireshark-37284-37285	No Patch	-	Plausible‡	Plausible	1482	11.5m	SL†

Table 2.1: Kali Experimental Results

2.8 Kali

Inspired by the fact that many patches generated by GenProg, AE, and RSRepair are semantically equivalent to eliminating functionalities, we build a new patch generation system, Kali. The basic idea behind Kali is to search a simple patch space that consists solely of patches that remove functionality. There are two potential goals: 1) if the correct patch simply removes functionality, find the patch, 2) if the correct patch does not simply remove functionality, generate a patch that modifies the functionality containing the defect. If the latter case occurs, the generated patch may nevertheless help developers to diagnose the defect.

For an existing statement, Kali deploys the following kinds of patches:

- **Redirect Branch:** If the existing statement is a branch statement, set the condition to true or false. The effect is that the then or else branch always executes.

- **Insert Return:** Insert a return before the existing statement. If the function returns a pointer, the inserted return statement returns NULL. If the function returns an integer, Kali generates two patches: one that returns 0 and another that returns -1.

- **Remove Statement:** Remove the existing statement. If the statement is a compound statement, Kali will remove all substatements inside it as well.

Statement Ordering: Each Kali patch targets a statement. Kali uses instrumented executions to collect information and order the executed statements as follows. Given a statement s and a test case i, $r(s, i)$ is the recorded execution counter that identifies the last execution of the statement s when the application runs with test case i. In particular, if the statement s is not executed at all when the application runs with the test case i, then $r(s, i) = 0$. Neg is the set of negative test cases (for which the unpatched application produces incorrect output) and Pos is the set of positive test cases (for which the unpatched application produces correct output). Kali computes three scores $a(s)$, $b(s)$, $c(s)$ for each statement s:

$$a(s) = |\ \{i \ \mid \ r(s,i) \neq 0,\ i \in \text{Neg}\}\ |$$
$$b(s) = |\ \{i \ \mid \ r(s,i) = 0,\ i \in \text{Pos}\}\ |$$
$$c(s) = \Sigma_{i \in \text{Neg}} r(s,i)$$

A statement s_1 has higher priority than a statement s_2 if $\text{prior}(s_1, s_2) = 1$, where prior is defined as:

$$prior(s_1, s_2) = \begin{cases} 1 & a(s_1) > a(s_2) \\ 1 & a(s_1) = a(s_2), b(s_1) > b(s_2) \\ 1 & \begin{array}{l} a(s_1) = a(s_2), b(s_1) = b(s_2), \\ c(s_1) > c(s_2) \end{array} \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, Kali prioritizes statements 1) that are executed with more negative test cases, 2) that are executed with less positive test cases, and 3) that are executed later during the executions with negative test cases. The Kali search space includes the top 500 ranked statements regardless of the file in which they appear.

Search: Kali deterministically searches the patch space in tiers: first all patches that change an if condition, then all patches that insert a return, then all patches that remove a statement. Within each tier, Kali applies the patch to the statements in the priority order identified above. It accepts a patch if the patch produces correct outputs for all of the inputs in the validation test suite.

2.8.1 Kali Evaluation Methodology

We evaluate Kali on all of the 105 defects in the GenProg set of benchmark defects [5]. We also use the validation test suites from this benchmark set. Our patch evaluation infrastructure is derived from the GenProg patch evaluation infrastructure [5]. For each defect, Kali runs its automatic patch generation and search algorithm to generate a sequence of candidate patches. For each candidate patch, Kali applies the patch to the application, recompiles the application, and uses the patch evaluation infrastructure to run the patched application on the inputs in the patch validation test suite. To check if the patch corrects the known incorrect behavior from the test suite, Kali first runs the negative test cases. To check if the patch preserves known correct behavior from the test suite, Kali next runs the positive test cases. If all of the test cases produce the correct output, Kali accepts the patch. Otherwise it stops the evaluation of the candidate patch at the first incorrect test case and moves on to evaluate the next patch.

Kali evaluates the php patches using the modified php test harness described in Section 2.3. It evaluates the gmp patches using a modified gmp test script that checks that all output components are correct. It evaluates the libtiff patches with augmented test scripts that compare various elements of the libtiff output image files from the patched executions with the corresponding elements from the correct image files. Other components of the image files change nondeterministically without affecting the correctness. The libtiff test scripts therefore do not fully check for correct outputs. After Kali obtains patches that pass the modified libtiff test scripts, we manually evaluate the outputs to filter all Kali patches that do not produce correct outputs for all of the inputs in the validation test suite. This manual evaluation rejects 7 libtiff patches, leaving only 5 plausible patches. Effective image comparison software would enable Kali to fully automate the libtiff patch evaluation.

We perform all of our Kali experiments (except for the fbc defects) on Amazon EC2 Intel Xeon 2.6GHz Machines running Ubuntu-64bit 14.04. The fbc application only runs in 32-bit environment, so we use a virtual machine with Intel Core 2.7GHz running Ubuntu-32bit 14.04 for fbc.

2.8.2 Experimental Results

Figure 2.1 presents the experimental results from our analysis of these patches. The figure contains a row for each defect for which at least one system (GenProg, RSRepair, AE, or Kali) generates a plausible patch. The second to fifth columns present the results of GenProg, RSRepair, AE, and Kali on each defect. "Correct" indicates that the system generates at least one correct patch for the defect. "Plausible" indicates that the system generates at least one plausible patch but no correct patches for the defect. "Implausible" indicates that all patches generated by the system for the defect are not plausible. "No Patch" indicates that the system does not generate any patch for the defect. "-" indicates that the RSRepair researchers chose not to include the defect in their study [82]. "‡" indicates that at least one of analyzed patches is not equivalent to a single functionality elimination modification.

Our results show that for the defects in the GenProg benchmark set, Kali generates correct patches for at least as many defects (3 for Kali vs. 3 for AE and 2 for GenProg and RSRepair) and plausible patches for at least as many defects (27 for Kali vs. 18 for GenProg, 10 for RSRepair, and 27 for AE).

Search Space and Time Results: The sixth column of Figure 2.1 presents the size of the search space for each defect (which is always less than 1700 patches). The seventh column presents the search times. Kali typically finds the patches in tens of minutes. If the search space does not contain a plausible patch, Kali typically searches the entire space in several hours and always less than seven hours.

It is not possible to directly compare the reported performance numbers for GenProg, RSRepair, and AE [55, 82, 104] with the numbers in Figure 2.1. First, the reported aggregate results for these prior systems include large numbers of implausible patches. The reported results for individual defects ([82], Table 2) report too few test case executions to validate plausible patches for the validation test suite (specifically, the reported number of test case executions is less than the number of test cases in the test suite). Second, these prior systems reduce the search space by requiring the developer to identify a target source code file to attempt to patch (Kali, of course, works with the entire application). Nevertheless, the search space sizes for these prior systems appear to be in the tens of thousands ([104], Table I) as opposed to hundreds for Kali. These numbers are consistent with the simpler Kali search space induced by the simpler set of Kali functionality deletion modifications.

Patch Classification: The last column of Figure 2.1 presents our classification of the Kali patches. "C" indicates that the Kali patch is correct. There are three defects for which Kali generates a correct patch. For two of the defects (php-bug-309111-309159, python-bug-69783-69784) both the Kali and developer patch simply delete an if statement. For php-bug-311346-311348, the Kali patch is a then redirect patch. The developer patch changes the else branch, but when the condition is true, the then branch and modified else branch have the same semantics.

"SL" indicates that the Kali and corresponding developer patches modify the same line of code. "*" indicates that the developer patch modified only the function that the Kali patch modified. "†" indicates that the developer patch modified other code outside the function. In many cases the Kali patch cleanly identifies the exact functionality and location that the developer patch modifies. Examples include changing the same if condition (fbc-bug-5458-5459, libtiff-bug-d13be72c-ccadf48a), changing the condition of an if statement when the developer patch modifies the then and/or else clause of that same if statement (python-bug-70098-70101, libtiff-bug-0860361d-1ba75257, wireshark-bug-37112-37111), deleting code that the developer patch encloses in an if statement (lighttpd-bug-1913-1914, php-bug-310673-310681, and deleting the same code (php-bug-308525-308529, libtiff-bug-0860361d-1ba75257, libtiff-bug-90d136e4-4c66680f, wireshark-bug-37172-37171, wireshark-bug-37172-37173, wireshark-bug-37284-37285) as the developer patch. Many of the patches correspond quite closely to the developer patch and move the application in the same direction.

"SF" indicates that the Kali and corresponding developer patches modify the same function. The number in parentheses is the distance in lines of code between the Kali patch and developer modifications. The Kali and developer patches typically modify common functionality and variables. Examples include reference counting (php-bug-309986-310009), caching (lighttpd-bug-1806-1807), and file encoding mechanism functionality (lighttpd-bug-2330-2331).

"DP" indicates that the Kali and developer patches modify different functions, but there is some dependence that connects the Kali and developer patches. Examples include changing the return value of a function invoked by code that the developer patch modifies (gmp-bug-14166-14167), deleting a call to a function that the developer patch modifies (php-bug-307931-307934), modifying memory management code for

```
 1  -if (y < 1000) {
 2  +if (y < 1000 && !1) {
 3     PyObject *accept = PyDict_GetItemString(
 4       moddict, "accept2dyear");
 5     if (accept != NULL) {
 6       int acceptval=PyObject_IsTrue(accept);
 7       if (acceptval == -1)
 8         return 0;
 9       if (acceptval) {
10         if (0 <= y && y < 69)
11           y += 2000;
12         else if (69 <= y && y < 100)
13           y += 1900;
14         else {
15           PyErr_SetString(PyExc_ValueError,
16             "year out of range");
17           return 0;
18         }
19         if (PyErr_WarnEx(
20                 PyExc_DeprecationWarning,
21   "Century info guessed for a 2-digit year.",
22                 1) != 0)
23           return 0;
24         }
25     }
26     else
27       return 0;
28   }
29   p->tm_year = y - 1900;
30   p->tm_mon--;
31   p->tm_wday = (p->tm_wday + 1) % 7;
```

Figure 2-17: Kali Patch for python-bug-69783-69784. Modify Lines 1-2.

the same data structure (php-bug-310370-310389), and accessing the same value, with either the Kali or the developer patch changing the value (lighttpd-bug-2661-2662, libtiff-bug-5b02179-3dfb33b).

The Kali patch for lighttpd-bug-1794-1795 (like the GenProg and AE patches) is an outlier — it deletes error handling code automatically generated by the yacc parser generator. The developer patch changes the yacc code to handle new configuration parameters. We do not see any of the automatically generated patches as providing useful information about the defect.

```
1   if (offset >= s1_len) {
2     php_error_docref(NULL TSRMLS_CC,
3       E_WARNING,
4       "The start position cannot " +
5       "exceed initial string length");
6     RETURN_FALSE;
7   }
8  -if (len > s1_len - offset) {
9  +if (len > s1_len - offset && !1)
10    len = s1_len - offset;
11  }
12  cmp_len = (uint) (len ? len : MAX(
13     s2_len, (s1_len - offset)));
```

Figure 2-18: Kali Patch for php-bug-309892-309910. Modify Lines 1-2.

```
1  -if (ctx->buf.len) {
2  +if ((ctx->buf.len) || 1) {
3     smart_str_appendl(&ctx->result,
4       ctx->buf.c, ctx->buf.len);
5     smart_str_appendl(&ctx->result,
6       output, output_len);
7     *handled_output = ctx->result.c;
8     *handled_output_len =
9       ctx->buf.len + output_len;
10    ctx->result.c = NULL;
11    ctx->result.len = 0;
12    smart_str_free(&ctx->buf);
13  } else {
14    *handled_output = NULL;
15  }
```

Figure 2-19: Kali Patch for php-bug-311346-311348. Modify lines 8-9.

python-bug-69783-69784: Figure 2-17 presents the Kali patch for python-bug-69783-69784. Like the GenProg, RSRepair, and AE patches, the patch for this defect deletes the if statement that implements two-digit years. Note that unlike these previous systems, which generate preprocessed code, Kali operates directly on and therefore preserves the structure of the original source code. To implement the deletion, Kali conjoins false (i.e., !1) to the condition of the if statement.

php-bug-309892-309910: Figure 2-18 presents the Kali patch for php-bug-309892-309910. Like the GenProg, RSRepair, and AE patches, this patch deletes the if statement that implements the obsolete check.

```
 1     if (ctx->buf.len) {
 2       smart_str_appendl(&ctx->result,
 3         ctx->buf.c, ctx->buf.len);
 4       smart_str_appendl(&ctx->result,
 5         output, output_len);
 6       *handled_output = ctx->result.c;
 7       *handled_output_len =
 8         ctx->buf.len + output_len;
 9       ctx->result.c = NULL;
10       ctx->result.len = 0;
11       smart_str_free(&ctx->buf);
12     } else {
13   -   *handled_output = NULL;
14   +   *handled_output = estrndup(output,
15   +      *handled_output_len = output_len);
16     }
```

Figure 2-20: Developer Patch for php-bug-311346-311348. Modify Lines 1-2.

php-bug-311346-311348: Figure 2-19 presents the Kali patch for php-bug-311346-311348. This code concatenates two strings, ctx->buf.c and output. The original code incorrectly set the result handled_output to NULL when the first string is empty. The Kali patch, in effect, deletes the else branch of the if statement on line 1 so that handled_output is correctly set when ctx->buf.c is empty and output is not empty. Figure 2-20 presents the developer patch. The developer patches the else branch to correctly set handled_output when ctx->buf.c is empty. The two patches have the same semantics.

2.9 Discussion

Our analysis substantially changed our understanding of the capabilities of the analyzed automatic patch generation systems. The majority of the reported GenProg, RSRepair, and AE patches in the relevant papers are not plausible. The overwhelming majority of these patches are incorrect and many of them are semantically equivalent to functionality deletion. In fact, Kali, a patch generation system that only deletes statements, can generate as many correct and plausible patches as GenProg, RSRepair, and AE do.

The results presented in this chapter highlight several important design considerations for generate-and-validate patch generation systems. First, such systems should not use weak proxies (such as the exit code of the program) — they should instead actually check that the patched program produces acceptable output. Second, the search space and search algorithm are critical — a successful system should use 1) a search space that contains successful patches and 2) a search algorithm that can search the space efficiently enough to find successful patches in an acceptable amount of time. Third, simply producing correct results on a validation test suite is (at least with current test suites) far from enough to ensure acceptable patches. Especially when the test suite does not contain test cases that protect desired functionality, unsuccessful patches can easily generate correct outputs.

Given this backdrop, what can one realistically expect from automatic patch generation systems moving forward? Currently available evidence indicates that improvements will require both 1) the use of richer search spaces with more correct patches and 2) the use of more effective search algorithms that can search the space more efficiently.

Perhaps most importantly, our results highlight important differences between machine-generated and human-generated patches. Even the plausible GenProg, AE, and RSRepair patches are overwhelming incorrect and simply remove functionality. The human-generated patches for the same defects, in contrast, are typically correct and usually modify or introduce new program logic. This result indicates that information other than simply passing a validation test suite is (at least with current test suites) important for producing correct patches.

Learning From Successful Patches: One way to obtain additional information is to learn from successful human patches. The remaining of this thesis will present techniques to learn universal properties and patching strategies of successful human patches. Chapter 4 presents Prophet, the first patch generation system that learns from human patches. Prophet analyzes a large database of revision changes extracted from open source project repositories to automatically learn features of successful patches. It then uses these features to recognize and prioritize correct patches within a larger space of candidate patches. On the GenProg benchmark set, Prophet generates correct patches for 18 defects (16 more than GenProg and 15 more than AE).

Chapter 6 presents Genesis, the first patch generation system that automatically infers productive code transforms and search spaces from human patches. Genesis operates on Java programs. On a systematically collected benchmark set of 49 errors, Genesis generates correct patches for 24 out of the 49 errors, while PAR, the previous state-of-the-art patch generation system for Java, generates correct patches for 11 out of the 49.

Automatic Code Transfer: Another way to obtain correct code is to obtain it from another application. Working with an input that exposes a potential security vulnerability, CodePhage searches an application database to automatically locate and transfer code that eliminates the vulnerability [94]. CodePhage successfully repaired 10 defects in 7 recipient applications via code transfer from 5 donor applications.

Learned Invariants: Successful executions are yet another source of useful information. Learning data structure consistency specifications from successful executions can enable successful data structure repair [34]. ClearView [80] observes successful executions to dynamically learn and enforce invariants that characterize successful executions. ClearView automatically generates successful patches that eliminate security vulnerabilities in 9 of 10 evaluated defects [80].

Targeted Patch Generation: Another source of information is to identify a specific set of defects and apply techniques that target that set of defects. Researchers have successfully targeted out of bounds accesses [25, 71, 90], null pointer dereferences [35, 61], divide by zero errors [61], memory leaks [43, 74], infinite loops [28, 50, 52], and integer and buffer overflows [95]. For the defects in scope, a targeted technique tends to generate patches with better quality than a search-based technique. For example, RCV [61], a recovery tool for divide-by-zero and null-dereference defects, successfully enables applications to recover from the majority of the systematically collected 18 CVE defects so that they exhibit identical behavior as the developer-patched application.

Specifications: Specifications, when available, can enable patch generation systems to produce patches that are guaranteed to be correct. AutoFix-E produces semantically sound candidate bug patches with the aid of a design-by-contract programming language (Eiffel) [79]. CodeHint uses partial specifications to automatically synthesize code fragments with the specified behavior [42]. Data structure repair techniques detect the inconsistency between a data structure state and a set of specified model constraints and enforce the violated constraints to repair the data structure state [32].

Chapter 3

A General Framework for Generate-and-Validate Systems

This chapter presents a general framework for generate-and-validate patch generation systems. This general framework covers all of the four patch generation systems described in this dissertation, Kali, SPR, Prophet and Genesis. Kali (see Section 2.8) is a patch generation system that only eliminates functionality. We develop Kali only for comparison with GenProg, AE, and RSRepair and do not advocate using Kali for patch generation tasks. SPR (see Section 4.2) is a patch generation system we built for C that forms the foundation for Prophet. Prophet (see Chapter 4 and Chapter 5) is a novel patch generation system which automatically learns from past successful human patches to prioritize the search of potentially correct patches. Genesis (see Chapter 6 and Chapter 7) is a novel patch generation system for Java. Like Prophet, Genesis learns from past successful human patches to prioritize correct patches. In addition, Genesis also automatically infers code transforms from the human patches to generate its patch generation search space.

Figure 3-1 presents the high level patch generation workflow. This workflow covers all four of the patch generation systems. In general, given a program that contains a defect and a set of test cases, at least one of which exposes the defect, a generate-and-validate system generates patches for the program with the following four steps: "run

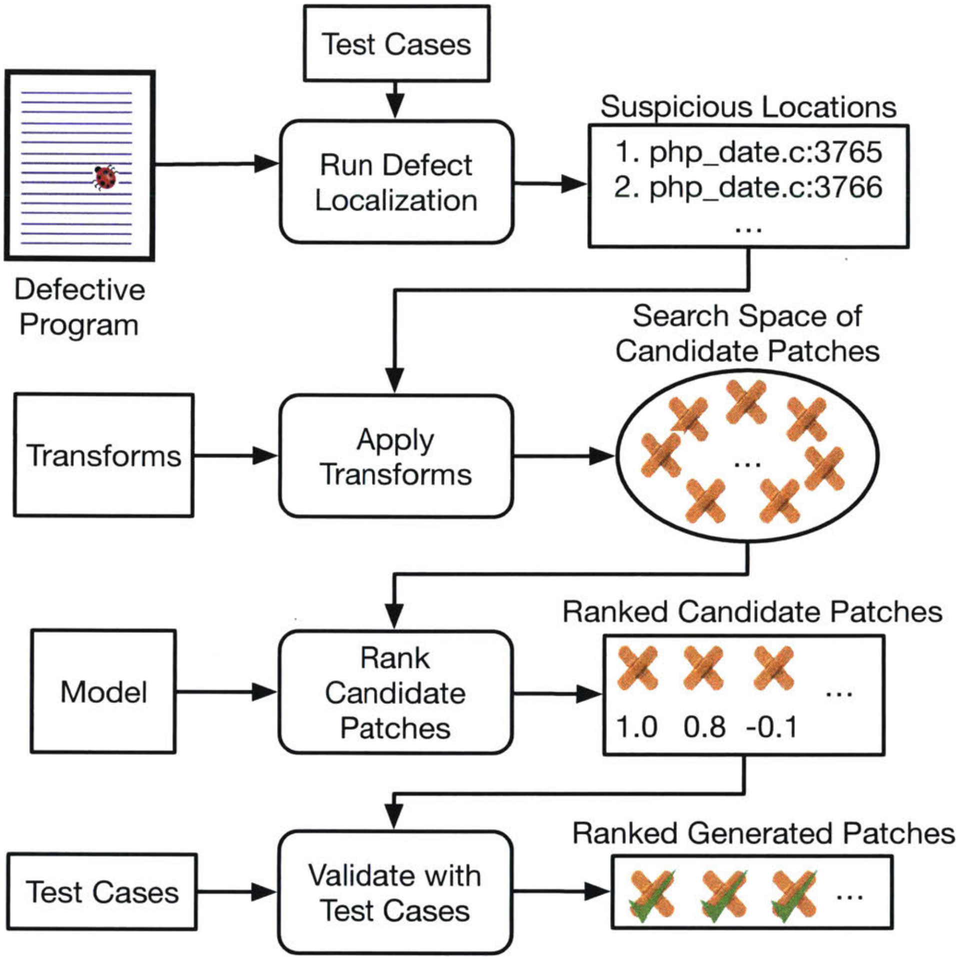


Figure 3-1: The Patch Generation Workflow of Generate-and-Validate Systems

defect localization", "apply transforms", "rank candidate patches", and "validate with test cases" as shown in Figure 3-1.

3.1 Run Defect Localiation

Defect localization is a standard technique designed to pinpoint the root cause of a software defect [47, 70, 78, 105]. It was originally designed to help human developers and has a wide range of applications on different software engineering tasks [47, 70, 78, 105]. Here we use it to direct the attention of the generate-and-validate system to suspicious program locations such as lines and statements. The generate-and-validate system first runs a defect localization algorithm on the program with the supplied test cases. The algorithm analyzes the execution traces of the program to produce a ranked list of suspicious program locations that are relevant to the root cause of the defect. Note that the defect localization is typically independent from the rest of the generate-and-validate system, i.e., the system could work with any defect localization technique.

Kali, SPR, and Prophet use a standard spectrum-based localization algorithm that prioritizes program statements 1) that are executed with more negative test cases, 2) that are executed with fewer positive test cases, and 3) that are executed later during executions with negative test cases. See Section 2.8 or Section 4.4.1 for more details. Genesis uses a stack-trace-based localization algorithm because the current version of Genesis focuses on three kinds of Java exception defects. The stack-trace-based algorithm performs better than the spectrum-based algorithm for those defects. See Section 6.3.3 for the defect localization algorithm in Genesis.

3.2 Apply Transforms

For each identified suspicious program location, the generate-and-validate system applies a set of transforms at the location to generate candidate patches. In the end, this step produces a search space of candidate patches that the system considers.

Note that there is an inherent trade-off between the coverage and tractability of the candidate patch search space. For a patch generation system to generate a correct patch for a defect, on one hand, the correct patch has to be inside the search space of the system. On the other hand, the search space has to be small enough so that the patch generation system can efficiently explore the space to find the correct patches. Kali, SPR, Prophet, and Genesis apply different kinds of transforms to generate their search spaces as follows.

- **Kali:** Kali applies three manually defined transforms. These transforms remove statements, remove branches, and skip computations at suspicious locations. All of these transforms are designed solely to eliminate functionality in the program. See Section 2.8 for more details.

- **Prophet and SPR:** Prophet and SPR use seven predefined transformation schemas to modify each suspicious location. The seven transformation schemas are "Condition Tightening", "Condition Loosening", "Condition Introduction", "Conditional Control Flow Introduction", "Insert Initialization", "Value Replacement", and "Copy and Replace". Each of these transformation schemas encodes one kind of modification that may be useful for fixing common development mistakes, such as missing a clause in a branch condition or missing an initialization statement. We manually developed these transformation schemas based on previous patch generation literature [31, 55] and my own experience of analyzing software defects. See Section 4.2 for the Prophet and SPR transformation schemas.

- **Genesis:** Unlike any other patch generation system, Genesis does not use predefined manual rules to obtain the transforms. Genesis, in contrast, automatically infers a set of code transforms from training human patches. In our experiments, Genesis infers in total 85 code transforms for null pointer deference, out of bound, and class cast defects. In comparison with manual transforms, each of the inferred transforms is more specific. But because there are so many more inferred transforms together they can more effectively navigate the inherent tradeoff between coverage and tractability. Our results show that the inferred transforms enable Genesis to generate correct patches for approximately two times more defects than manually defined transforms in PAR [49], a previous patch generation system for Java. See Section 6.2 for the code transform inference technique. See Section 7.4 for examples of the inferred code transforms in Genesis.

3.3 Rank Candidate Patches

Given the generated search space, the generate-and-validate system ranks the candidate patches in the space to determine the search order. The goal of this step is to prioritize potentially correct patches in the patch generation process. This ranking enables the system to produce more correct patches within a time limit. It also enables the system to rank correct patches ahead of plausible but incorrect patches (i.e., patches that pass the test cases but produce incorrect output for other inputs) in the generated patch list and therefore makes the patch generation results more useful for the developer.

- **Kali:** Kali uses a manual rule to rank candidate patches. It first prioritizes patches that remove branches, then prioritizes patches that skip computations, and finally considers patches that remove individual statements.

- **SPR:** SPR uses an elaborate set of manual heuristic rules to rank candidate patches generated by the seven transformation schemas. The rules are highly tuned and in general tend to prioritize patches that manipulate conditions. See Section 4.2.4 for the ranking rules in SPR.

- **Prophet and Genesis:** Prophet and Genesis use a machine learning technique to rank each candidate patch in the space. The goal of this step is to prioritize potentially correct patches in the patch generation process. The learning technique operates with a probabilistic model to identify universal code properties that correlate with successful human patches in a training set. Once trained, the probabilistic model assigns a probability score to each candidate patch. This score indicates the likelihood that the patch is correct. Prophet and Genesis sort all candidate patches in the space according to their scores. In our experiments, the learned algorithm enables Prophet to rank correct patches as the first generated patch for five more C defects than the manual rules in SPR. The learned algorithm also enables Genesis to generate correct patches for two more Java defects. See Section 4.3 for the learning technique in Prophet. See Section 6.3.1 for the implementation of the learning technique in Genesis.

3.4 Validate with Test Cases

The generate-and-validate system finally validates each candidate patch one by one in the ranked order against the supplied test cases. The system returns an ordered sequence of patches that validate (i.e., produce correct outputs for all test cases in the test suite) as the result of the patch generation process.

Prophet, SPR, and Genesis implement a condition synthesis technique that can prune away invalid patches and speed up the validation process. Many candidate patches in Prophet, SPR, and Genesis manipulate branch conditions. If two such patches modify the branch condition and trigger the same sequence of branch direction combinations on a test case, these two patches will produce the same output. Therefore the condition synthesis technique groups such patches together. If one of those patches fails on the test case, then a patch generation system can discard the rest of the patches in the same group.

There are two ways to implement the condition synthesis techniques. Prophet and SPR explicitly search different branch direction combinations and then synthesize concrete patches only for those combinations that enable the program to pass all test cases. See Section 4.2.3 for the condition synthesis algorithm in Prophet and SPR. Genesis instruments the program to record, for each tested candidate patch, the evaluation results of the modified expressions (conditions). Genesis then prunes away any candidate patch that would produce the same value (branch direction) combinations as a previously tested patch. See Section 6.3.2 for the implementation of the condition synthesis algorithm in Genesis.

One intriguing result is that the condition synthesis technique prunes away less patches for Genesis than for Prophet and SPR. In fact, Genesis generates fewer correct patches when it enables the condition synthesis – the benefit of condition synthesis does not even offset the instrumentation overhead. One reason is that, in comparison with Prophet and SPR, Genesis works with a relatively large number of transforms and each of the transforms is smaller. The current condition synthesis technique do not group candidate patches from different transforms and therefore may miss pruning opportunities for the Genesis search space. Another reason is that the Genesis inferred transforms tend to generate more candidate patches that manipulate function calls. The condition synthesis technique cannot prune away such patches. See Section 7.5 for our experimental results about this phenomena.

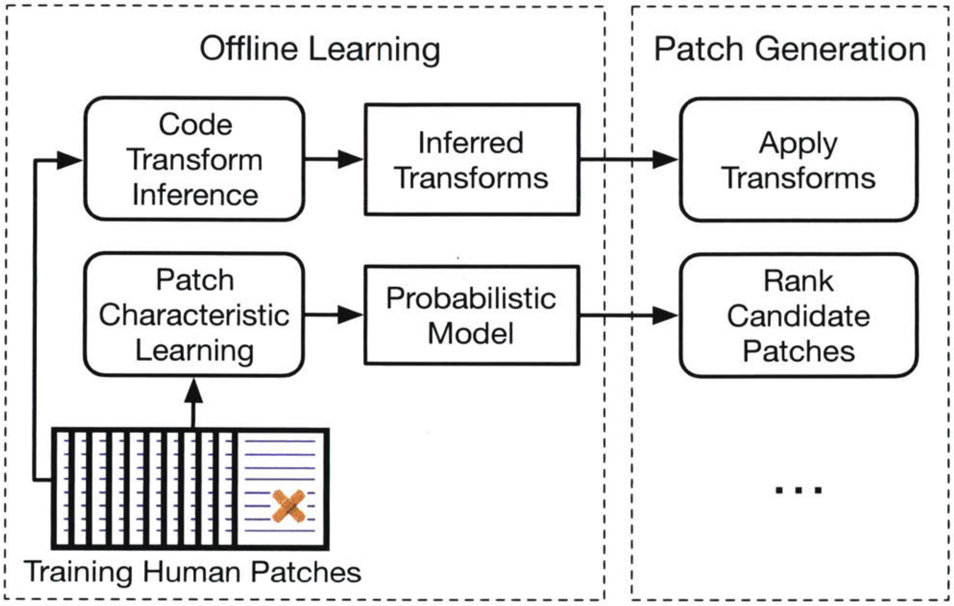


Figure 3-2: The Offline Learning of Prophet and Genesis

3.5 Offline Learning in Prophet and Genesis

Prophet and Genesis differ from previous patch generation systems in that Prophet and Genesis leverage information learned from past successful human patches to enhance the patch generation process. Figure 3-2 presents the offline learning phase of Prophet and Genesis. Prophet and Genesis operate with a database of training human patches. In this dissertation, we collect such training patches from open source repositories. Prophet and Genesis automatically learn a probabilistic model from the training human patches to predict the correctness of each candidate patch. During the patch generation phase, the systems apply the learned mode to rank candidate patches. Genesis further automatically infers code transforms that summarize patching strategies employed by human developers. During the patch generation phase, Genesis applies the inferred code transforms to generate a productive set of candidate patches instead of relying on manually crafted transforms.

Our experimental results in Chapter 5 and Chapter 7 show that by collectively leveraging development efforts embedded in the human patches, Prophet and Genesis outperform previous patch generation systems, generating more correct patches and ranking correct patches ahead for more defects. The results demonstrate that exploiting additional information learned from successful human patches is a very promising direction to build powerful generate-and-validate systems.

Chapter 4

Learning Universal Properties of Correct Code with Prophet

This chapter presents Prophet, a novel generate-and-validate patch generation system for repairing defects in large real-world C applications. Prophet works with a set of successful patches obtained from open-source software repositories to learn a probabilistic model of correct code. It uses this model to rank and identify correct patches within an automatically generated space of candidate patches. The goal of Prophet is to obtain a correct patch as the first (or one of the first few) patches to validate.

Universal Feature Extraction: During the offline learning phase, Prophet operates with a parameterized probabilistic model to capture useful code properties in the collected human patches. For each patch, Prophet extracts potentially useful patch properties as *universal features.* These features include *program value features*, which capture relationships between how variables and constants are used in the original program and how they are used in the patch, and *modification features*, which capture relationships between the kind of program modification that the patch applies and the kinds of statements that appear near the patched statement in the original program. Prophet converts the extracted features into a binary feature vector.

Each patch inserts new code into the program. But correctness does not depend only on the new code — it also depends on how that new code interacts with the application into which it is inserted. The learned correctness model therefore works with features that capture critical aspects of how the new code interacts with the surrounding code from the patched application.

Probabilistic Model: Prophet operates with a parameterized probabilistic model that, once the model parameters are determined, assigns a probability to each candidate patch in the search space. This probability indicates the likelihood that the patch is correct. The model is the product of a geometric distribution determined by the Prophet defect localization algorithm (which identifies target program statements for the patch to modify) and a log-linear distribution determined by the model parameters and the feature vector.

Maximum Likelihood Estimation: Given a training set of correct patches, Prophet learns the model parameters by maximizing the likelihood of observing the training set. The intuition behind this approach is that the learned model should assign a high probability to each of the correct patches in the training set.

Patch Ranking and Validation: Prophet extracts features from candidate patches in the search space and converts them into binary feature vectors. Prophet then uses the learned model and the extracted binary feature vectors to compute a probability score for each patch in the search space of candidate patches. Prophet then sorts the candidates according to their scores and validates the patches against the supplied test suite in that order. It returns an ordered sequence of patches that validate (i.e., produce correct outputs for all test cases in the test suite) as the result of the patch generation process.

Universal Roles and Program Value Features: A key challenge for Prophet is to identify, learn, and exploit universal properties of correct code. Many surface syntactic elements of the correct patches in the Prophet training set (such as variable names and types) tie the patches to their specific applications and prevent the patches from directly generalizing to other applications.

The Prophet program value features address this challenge as follows. Prophet uses a static analysis to obtain a set of application-independent *atomic characteristics* for each program value (i.e., variable or constant) that the patch manipulates. Each atomic characteristic captures a (universal, application-independent) role that the value plays in the original or patched program (for example, a value may occur in the condition of an if statement or be returned as the value of an enclosing function).

Prophet then defines program value features that capture relationships between the roles that the same value plays in the patch and the original code that the patch modifies. These relationships capture interactions between the patch and the patched code that correlate with patch correctness and incorrectness. Because the features are derived from universal, application-independent roles, they generalize across different applications.

Sources. The previous version of this research presented in this chapter appeared in [58]. Prophet is built on our previous patch generation system, SPR, which appeared in [59].

4.1 Motivating Example

We next present an example that illustrates how Prophet corrects a defect in the PHP interpreter. The PHP interpreter (before version 5.3.5 or svn version 308315) contains a defect (PHP bug #53971) in the Zend execution engine. If a PHP program accesses a string with an out-of-bounds offset, the PHP interpreter may produce spurious runtime errors even in situations where it should suppress such errors.

Figure 4-1 presents (simplified) code (from the source code file Zend/zend_execute.c) that contains the defect. The C function at line 1 in Figure 4-1 implements the read operation that fetches values from a container at a given offset. The function writes these values into the data structure referenced by the first argument (result).

When a PHP program accesses a string with an offset, the second argument (container_ptr) of this function references the accessed string. The third argument (dim) identifies the specified offset values. The code at lines 17-18 checks whether the specified offset is within the length of the string. If not, the PHP interpreter generates a runtime error indicating an offset into an uninitialized part of a string (lines 32-34).

In some situations PHP should suppress these out-of-bounds runtime errors. Consider, for example, a PHP program that calls isset(str[1000]). According to the PHP specification, this call should not trigger an uninitialized data error even if the length of the PHP string str is less than 1000. The purpose of isset() is to check if a value is properly set or not. Generating an error message when isset() calls the procedure in Figure 4-1 is invalid because it interferes with the proper operation of isset().

In such situations the last argument (type) at line 3 in Figure 4-1 is set to 3. But the implementation in Figure 4-1 does not properly check the value of this argument before generating an error. The result is spurious runtime errors and, depending on the PHP configuration, potential denial of service.

Offline Learning: Prophet works with a training set of successful human patches to obtain a probabilistic model that captures why these patches were successful. We obtain this training set by collecting revision changes from open source repositories. In our example, we train Prophet with patches from seven open source projects (apr, curl, httpd, libtiff, python, subversion, and wireshark). Although revision changes for PHP are available, we exclude these revision changes from this training set. During the offline learning phase, Prophet performs the following steps:

- **Extract Features:** For each patch in the training set, Prophet analyzes a structural diff on the abstract syntax trees of the original and patched code to extract both 1) modification features, which summarize how the patch modifies the program given characteristics of the surrounding code and 2) program value features, which summarize relationships between roles that values accessed by the patch play in the original unpatched program and in the patch.

- **Learn Model Parameters:** Prophet operates with a parameterized log-linear probabilistic model in which the model parameters can be interpreted as weights that capture the importance of different features. Prophet learns the model parameters via maximum likelihood estimation, i.e., the Prophet learning algorithm attempts to find parameter values that maximize the probability of observing the collected training set in the probabilistic model.

```c
static void zend_fetch_dimension_address_read(
    temp_variable *result, zval **container_ptr,
    zval *dim, int dim_type, int type)
{
  zval *container = *container_ptr;
  zval **retval;
  switch (Z_TYPE_P(container)) {
    ...
    case IS_STRING: {
      zval tmp;
      zval *ptr;
      ...
      ALLOC_ZVAL(ptr);
      INIT_PZVAL(ptr);
      Z_TYPE_P(ptr) = IS_STRING;

      if (Z_LVAL_P(dim) < 0 ||
          Z_STRLEN_P(container) <= Z_LVAL_P(dim)) {
//      A plausible but incorrect patch that validates
//      if (!(type == 3)) return;

//      An unconstrained patch with abstract condition C
//      if (C), where C is unconstrained
//      An partially instantiated patch
//      if (C), where C checks the variable "type"

//      The guard that the correct Prophet patch inserts
//      before the following error generation statement.
//      This Prophet patch is identical to the (correct)
//      developer patch.
//      if (!(type == 3))
          zend_error(E_NOTICE,
                    "Uninitialized string offset: %ld",
                    (*dim).value.lval);
        Z_STRVAL_P(ptr) = STR_EMPTY_ALLOC();
        Z_STRLEN_P(ptr) = 0;
      } else {
        Z_STRVAL_P(ptr) = (char*)emalloc(2);
        Z_STRVAL_P(ptr)[0] =
            Z_STRVAL_P(container)[Z_LVAL_P(dim)];
        Z_STRVAL_P(ptr)[1] = 0;
        Z_STRLEN_P(ptr) = 1;
      }
      AI_SET_PTR(result, ptr);
      return;
    } break;
    ...
  }
}
```

Figure 4-1: Simplified Code for PHP bug #53971

Apply Prophet: We apply Prophet to automatically generate a patch for this defect. Specifically, we provide Prophet with the PHP source code that contains the defect and a test suite that contains 6957 test cases. One of the test cases exposes the defect (i.e., the unpatched version of PHP produces incorrect output for this test case). The remaining 6956 test cases are to prevent regression (the unpatched version of PHP produces correct outputs for these test cases). Prophet generates a patch with the following steps:

- **Defect Localization:** Prophet first performs a dynamic analysis of the execution traces of the PHP interpreter on the supplied test suite to identify a set of candidate program points for the patch to modify. In our example, the Prophet defect localization algorithm observes that the negative test case executes the statement at lines 32-34 in Figure 4-1 while the positive test cases rarely execute this statement. Prophet therefore generates candidate patches that modify this statement (as well as candidate patches that modify other statements).

- **Search Space Generation:** Prophet works with a set of transformation schemas to generate candidate patches. Some (but by no means all) of these candidate patches are generated by a transformation schema (see lines 22-23) that adds an if statement to guard (conditionally execute) the statement at lines 32-34 in Figure 4-1. This transformation schema contains an abstract condition that the Prophet condition synthesis algorithm will eventually instantiate with a concrete condition.

  During search space generation and candidate patch ranking, Prophet does not attempt to fully instantiate the patch. It instead works with *partially instantiated patches* that identify the variable that the final concrete condition will check (but not the final concrete condition itself).

  In our example one of the partially instantiated patches is shown as lines 24-25. It 1) adds an if statement guard before the statement at lines 32-34 in Figure 4-1 (the statement that generates the error message) and 2) has a condition that checks the function parameter type.

- **Rank Candidate Patches:** Prophet computes a feature vector for each candidate (fully or partially instantiated) patch in the search space. It then applies the learned model to the computed feature vector to obtain a probability that the corresponding patch is correct. It then ranks the generated patches according to the computed correctness probabilities.

  In our example, the model assigns a relatively high correctness probability to the partially instantiated patch mentioned above (lines 32-34) because it has several features that positively correlate with correct patches in the training set. For example, 1) it adds an if statement to guard a call statement and 2) the guard condition checks a parameter of the enclosing procedure.

- **Validate Candidate Patches:** Prophet then uses the test suite to attempt to validate the candidate patches (including partially instantiated patches) in order of highest patch correctness probability. When the validation algorithm encounters a partially instantiated patch, Prophet invokes the Prophet condition synthesis algorithm to obtain concrete conditions that fully instantiate the patch (see Section 4.2.3). In our example, the condition synthesis algorithm comes back with the condition (type != 3) (the resulting patch appears at line 31 in Figure 4-1). This patch is the first patch to validate (i.e., it is the first generated patch that produces correct outputs for all of the test cases in the test suite).

The generated Prophet patch is correct and identical to the developer patch for this defect. Note that the Prophet search space may contain incorrect patches that nevertheless validate (because they produce correct outputs for all test cases in the test suite). In our example, line 20 in Figure 4-1 presents one such patch. This patch directly returns from the function if type != 3. This patch is incorrect because it does not properly set the result data structure (referenced by the result argument) before it returns from the function. Because the negative test case does not check this result data structure, this incorrect patch nevertheless validates. The Prophet

$$\begin{array}{lll}
c & := & \texttt{1} \mid \texttt{0} \mid c_1 \texttt{ \&\& } c_2 \mid c_1 \texttt{ || } c_2 \mid \texttt{!}c_1 \mid (c_1) \mid v\texttt{==}const \\
simps & := & v \texttt{ = } v_1 \texttt{ op } v_2 \mid v \texttt{ = } c \mid \texttt{print } v \mid v \texttt{ = read} \\
ifs & := & \texttt{if } (c)\ \ell_1\ \ell_2 \\
absts & := & \texttt{if } (c \texttt{ \&\& !abstc})\ \ell_1\ \ell_2 \mid \texttt{if } (c \texttt{ || abstc})\ \ell_1\ \ell_2 \\
s & := & \texttt{skip} \mid \texttt{stop} \mid simps \mid ifs \mid absts \\
\multicolumn{3}{l}{v,\ v_1,\ v_2 \in \texttt{Variable} \quad const \in \texttt{Int} \quad \ell_1,\ \ell_2 \in \texttt{Label}} \\
\multicolumn{3}{l}{c,\ c_1,\ c_2 \in \texttt{CondExpr} \quad s \in \texttt{Stmt} \quad ifs \in \texttt{IfStmt}} \\
\multicolumn{3}{l}{simps \in \texttt{SimpleStmt} \quad absts \in \texttt{AbstCondStmt}}
\end{array}$$

Figure 4-2: The Language for Illustrating Transformation Schemas

model ranks this plausible but incorrect patch below the correct patch because the incorrect patch inserts a return statement before a subsequent assignment statement in a code block. This interaction between the patch and the surrounding code incurs a significant penalty in the learned model.

4.2 Transformation Schemas and Condition Synthesis in Prophet and SPR

Prophet is built on our previous patch generation system, SPR. Prophet and SPR share the same search space and patch validation algorithm. The main difference between SPR and Prophet is that SPR uses a set of manual heuristic rules to determine the patch validation order, while Prophet uses a machine learning algorithm to prioritize potentially correct patches during the search.

This section presents the search space and the condition synthesis technique in Prophet and SPR. The Prophet and SPR search space is derived from a set of transformation schemas which target common mistakes developers would made in C programs. Prophet and SPR use a novel validation technique called condition synthesis which exploits the structure of the derived search space to efficiently prune away candidate patches that cannot pass test cases.

We use a simple imperative core language (Section 4.2.1) to present the key concepts in the Prophet transformation schemas and validation algorithm. Section 4.2.2 presents the transformation schemas as they apply to the core language. Section 4.2.3 presents the validation algorithm with the condition synthesis technique.

4.2.1 Core Language

Language Syntax: Figure 4-2 presents the syntax of the core language that we use to present our algorithm. A program in the language is defined as $\langle p, n\rangle$, where p : Label $\rightarrow$ Statement maps each label to the corresponding statement, n : Label $\rightarrow$ Label maps each label to the label of the next statement to execute in the program. ℓ_0 is the label of the first statement in the program.

The language in Figure 4-2 contains arithmetic statements and if statements. An if statement of the form "`if` (c) ℓ_1 ℓ_2" transfers the execution to ℓ_1 if c is 1 and transfers the execution to ℓ_2 if c is 0. The language uses if statements to encode loops. A statement of the form "v `= read`" reads an integer value from the input and stores the value to the variable v. A statement of the form "`print` v" prints the value of the variable v to the output. Conditional statements that contain an abstract condition `abstc` (i.e., `AbstCondStmt`) are temporary statements that the algorithm may introduce into a program during condition synthesis. Such statements do not appear in the original or patched programs.

Operational Semantics: A program state $\langle \ell, \sigma, I, O, D, R, S\rangle$ is composed of the current program point (a label ℓ), the current environment that maps each variable to its value (σ : Variable $\rightarrow$ Int), the remaining input (I), the generated output (O), a sequence of future abstract condition values (D), a sequence of recorded abstract condition values (R), and a sequence of recorded environments for each abstract condition execution (S). I and O are sequences of integer values (i.e., Sequence(Int)). D and R are sequences of zero or one values (i.e., Sequence(0 | 1)). S is a sequence of environments (i.e., Sequence(Variable $\rightarrow$ Int)).

Figure 4-3 presents the small step operational semantics of our language for statements without abstract conditions. "$\circ$" in Figure 4-3 is the sequence concatenation operator. The notation "$\sigma \vdash c \Rightarrow x$" indicates that the condition c evaluates to x

$$\frac{p(\ell) = \texttt{skip}}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle n(\ell), \sigma, I, O, D, R, S\rangle}$$

$$\frac{p(\ell) = \texttt{stop}}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \texttt{nil}, \sigma, I, O, D, R, S\rangle}$$

$$\frac{p(\ell) = v \ \texttt{=} \ \mathit{const}}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle n(\ell), \sigma[\mathrm{v}\mapsto \mathit{const}], I, O, D, R, S\rangle}$$

$$\frac{p(\ell) = v \ \texttt{=} \ v_1 \ \texttt{op} \ v_2 \qquad x = \sigma(v_1) \ \mathrm{op} \ \sigma(v_2)}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle n(\ell), \sigma[\mathrm{v}\mapsto x], I, O, D, R, S\rangle}$$

$$\frac{p(\ell) = v \ \texttt{=} \ \texttt{read} \qquad I = x \circ I\prime}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle n(\ell), \sigma[\mathrm{v}\mapsto x], I\prime, O, D, R, S\rangle}$$

$$\frac{p(\ell) = \texttt{print} \ v}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle n(\ell), \sigma, I, O \circ \sigma(v), D, R, S\rangle}$$

$$\frac{p(\ell) = \texttt{if} \ (c) \ \ell_1 \ \ell_2 \qquad \sigma \vdash c \Rightarrow 1}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_1, \sigma, I, O, D, R, S\rangle}$$

$$\frac{p(\ell) = \texttt{if} \ (c) \ \ell_1 \ \ell_2 \qquad \sigma \vdash c \Rightarrow 0}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_2, \sigma, I, O, D, R, S\rangle}$$

Figure 4-3: Small Step Operational Semantics for Statements without Abstract Condition

under the environment σ. D, R, and S are unchanged in these rules because these statements do not contain an abstract condition.

4.2.2 Transformation Schemas

Figure 4-4 presents our program transformation function M. It takes a program $\langle p, n\rangle$ and produces a set of candidate modified programs after transformation schema application. In Figure 4-4 *TargetL*($\langle p, n\rangle$) is the set of labels of target statements to transform. Our defect localization algorithm (Section 4.4.1) identifies this set of statements. *SimpleS*(p) denotes all simple statements (i.e. `SimpleStmt`) in p. *Vars*(p) and *Vars*(s) denote all variables in the program p and in the statement s, respectively. *Consts*(p) denotes all constants in p.

The program transformation function implements the following seven transformation schemas:

- **Condition Tightening** ($\mathrm{M}_{\mathbf{Tighten}}$)**:** Given a target if statement, the function transforms the condition of the if statement by conjoining an additional (synthesized) condition to the original if condition.

- **Condition Loosening** ($\mathrm{M}_{\mathbf{Loosen}}$)**:** Given a target if statement, the function transforms the condition of the if statement by disjoining an additional (synthesized) condition to the original if condition.

- **Condition Introduction** ($\mathrm{M}_{\mathbf{Guard}}$)**:** Given a target statement, the function transforms the program so that the statement executes only if a (synthesized) condition is true.

- **Conditional Control Flow Introduction** ($\mathrm{M}_{\mathbf{Control}}$)**:** The function inserts a new control flow statement (return, break, or goto an existing label) that executes only if a (synthesized) condition is true.

- **Insert Initialization** ($\mathrm{M}_{\mathbf{Init}}$)**:** Given a target statement, the function generates repairs that insert a memory initialization statement before the identified statement.

$$
\begin{array}{lcl}
\mathrm{M}(\langle p,n\rangle) & = & \mathrm{M}_{\mathrm{IfStmt}}(\langle p,n\rangle) \cup \mathrm{M}_{\mathrm{Stmt}}(\langle p,n\rangle) \\
\mathrm{M}_{\mathrm{IfStmt}}(\langle p,n\rangle) & = & \cup_{\ell\in \mathit{TargetL}(\langle p,n\rangle), p(\ell)\in \mathrm{IfStmt}}(\mathrm{M}_{\mathrm{Tighten}}(\langle p,n\rangle,\ell) \cup \mathrm{M}_{\mathrm{Loosen}}(\langle p,n\rangle,\ell)) \\
\mathrm{M}_{\mathrm{Stmt}}(\langle p,n\rangle) & = & \cup_{\ell\in \mathit{TargetL}(\langle p,n\rangle)}(\mathrm{M}_{\mathrm{Control}}(\langle p,n\rangle,\ell) \cup \mathrm{M}_{\mathrm{Init}}(\langle p,n\rangle,\ell) \cup \mathrm{M}_{\mathrm{Guard}}(\langle p,n\rangle,\ell) \cup \mathrm{M}_{\mathrm{Rep}}(\langle p,n\rangle,\ell) \cup \mathrm{M}_{\mathrm{CopyRep}}(\langle p,n\rangle,\ell)) \\
\mathrm{M}_{\mathrm{Tighten}}(\langle p,n\rangle,\ell) & = & \{\langle p[\ell \mapsto \mathrm{if}\ (c\ \texttt{\&\& !abstc})\ \ell_1\ \ell_2], n\rangle\},\ \text{where } p(\ell) = \mathrm{if}\ (c)\ \ell_1\ \ell_2 \\
\mathrm{M}_{\mathrm{Loosen}}(\langle p,n\rangle,\ell) & = & \{\langle p[\ell \mapsto \mathrm{if}\ (c\ \texttt{|| abstc})\ \ell_1\ \ell_2], n\rangle\},\ \text{where } p(\ell) = \mathrm{if}\ (c)\ \ell_1\ \ell_2 \\
\mathrm{M}_{\mathrm{Control}}(\langle p,n\rangle,\ell) & = & \{\langle p[\ell' \mapsto p(\ell)][\ell'' \mapsto \texttt{stop}][\ell \mapsto \mathrm{if}\ (0\ \texttt{|| abstc})\ \ell''\ n(\ell)], n[\ell' \mapsto n(\ell)][\ell \mapsto \ell'][\ell'' \mapsto \ell']\rangle\} \\
\mathrm{M}_{\mathrm{Guard}}(\langle p,n\rangle,\ell) & = & \{\langle p[\ell' \mapsto p(\ell)][\ell \mapsto \texttt{if (1 \&\& !abstc)}\ \ell'\ n(\ell)], n[\ell' \mapsto n(\ell)]\rangle\} \\
\mathrm{M}_{\mathrm{Init}}(\langle p,n\rangle,\ell) & = & \{\langle p[\ell' \mapsto p(\ell)][\ell \mapsto v\ \texttt{=}\ 0], n[\ell' \mapsto n(\ell)][\ell \mapsto \ell']\rangle \mid \forall v \in \mathit{Vars}(p(\ell))\} \\
\mathrm{M}_{\mathrm{Rep}}(\langle p,n\rangle,\ell) & = & \{\langle p[\ell \mapsto s], n\rangle \mid s \in \mathrm{RepS}(p, p(\ell))\} \\
\mathrm{M}_{\mathrm{CopyRep}}(\langle p,n\rangle,\ell) & = & \{\langle p[\ell' \mapsto p(\ell)][\ell \mapsto s], n[\ell' \mapsto n(\ell)][\ell \mapsto \ell']\rangle, \\
 & & \langle p[\ell' \mapsto p(\ell)][\ell \mapsto s')], n[\ell' \mapsto n(\ell)][\ell \mapsto \ell']\rangle \mid \forall s \in \mathit{SimpleS}(\langle p,n\rangle), \forall s' \in \mathrm{RepS(p,s)}\} \\
\mathrm{RepS}(p, v\ \texttt{=}\ v_1\ \texttt{op}\ v_2) & = & \{v'\ \texttt{=}\ v_1\ \texttt{op}\ v_2, v\ \texttt{=}\ v'\ \texttt{op}\ v_2, v\ \texttt{=}\ v_1\ \texttt{op}\ v' \mid \forall v' \in \mathit{Vars}(p)\} \\
\mathrm{RepS}(p, v\ \texttt{=}\ \mathit{const}) & = & \{v'\ \texttt{=}\ \mathit{const}, v\ \texttt{=}\ \mathit{const'} \mid \forall v' \in \mathit{Vars}(p), \forall \mathit{const'} \in \mathit{Consts}(p)\} \\
\mathrm{RepS}(p, v\ \texttt{=}\ \texttt{read}) & = & \{v'\ \texttt{=}\ \texttt{read} \mid \forall v' \in \mathit{Vars}(p)\} \\
\mathrm{RepS}(p, \texttt{print}\ v) & = & \{\texttt{print}\ v' \mid \forall v' \in \mathit{Vars}(p)\} \\
\mathrm{RepS}(p, s) & = & \emptyset,\ \text{where } s \notin \texttt{SimpleStmt}
\end{array}
$$

ℓ' and ℓ'' are fresh labels

Figure 4-4: The Program Transformation Function M

- **Value Replacement ($\mathrm{M}_{\mathbf{Rep}}$):** Given a target statement, the function generates patches that either 1) replace one variable with another, 2) replace an invoked function with another function, or 3) replace a constant with another constant.

- **Copy and Replace ($\mathrm{M}_{\mathbf{CopyRep}}$):** Given a target statement, the function generates repairs that copy an existing statement to the program point before the target statement and then apply a Value Replacement transformation.

Note that the first four schemas introduce an abstract condition into the generated candidate programs that will be handled by condition synthesis. Also note that $\mathrm{RepS}(p, s)$ is an utility function that returns the set of statements generated by replacing a variable or a constant in s with other variables or constants in p.

4.2.3 Validation Algorithm with Condition Synthesis

Figure 4-5 presents our main validation algorithm with condition synthesis. Given a candidate program $\langle p\prime, n\prime \rangle$ that may contain an abstract condition, a set of positive test cases $PosT$, and a set of negative test cases $NegT$, the algorithm produces a patched program $\langle p\prime, n\prime \rangle$ that passes all test cases. $\mathrm{Exec}(\langle p, n \rangle, I, D)$ at lines 12 and 20 produces the results of running the program $\langle p, n \rangle$ on the input I given the future abstract condition value sequence D. $\mathrm{Test}(\langle p, n \rangle, NegT, PosT)$ at lines 28 and 34 produces a boolean to indicate whether the program $\langle p, n \rangle$ passes all test cases. See Figure 4-6 for relevant definitions.

If the candidate program does not contain an abstract condition, the algorithm simply uses Test to validate the program with test cases (lines 35-36). Otherwise the algorithm applies condition synthesis in two stages, condition value search and condition generation.

Condition Value Search: We augment the operational semantics in Figure 4-3 to handle statements with an abstract condition. Figure 4-7 presents the additional rules. The first two rules specify the case where the result of the condition does not depend on the abstract condition (the semantics implements short-circuit conditionals). In

```
Input  : a candidate program $\langle p', n' \rangle$ that may contain an abstract condition
Input  : positive and negative test cases $NegT$ and $PosT$, each is a set of pairs
         $\langle I, O \rangle$ where $I$ is the test input and $O$ is the expected output.
Output : the patched program or $\emptyset$ if the validation failed
if $p'$ contains abstc then
    $R' \longleftarrow \epsilon$
    $S' \longleftarrow \epsilon$
    for $\langle I, O \rangle$ in $NegT$ do
        $\langle O', R, S \rangle \longleftarrow \mathrm{Exec}(\langle p', n' \rangle, I, \epsilon)$
        $cnt \longleftarrow 0$
        while $O' \neq O$ and $cnt \leq 10$ do
            if $cnt = 10$ then
                $D \longleftarrow 1 \circ 1 \circ 1 \circ 1 \cdots$
            else
                $D \longleftarrow \mathrm{Flip}(R)$
            $\langle O', R, S \rangle \longleftarrow \mathrm{Exec}(\langle p', n' \rangle, I, D)$
            $cnt \longleftarrow cnt + 1$
        if $O \neq O'$ then
            skip to the next candidate $\langle p', n' \rangle$
        else
            $R' \longleftarrow R' \circ R$
            $S' \longleftarrow S' \circ S$
    for $\langle I, O \rangle$ in $PosT$ do
        $\langle O', R, S \rangle \longleftarrow \mathrm{Exec}(\langle p', n' \rangle, I, \epsilon)$
        $R' \longleftarrow R' \circ R$
        $S' \longleftarrow S' \circ S$
    $C \longleftarrow \{\}$
    for $\sigma$ in $S'$ do
        $C \longleftarrow C \cup \{(v \texttt{ == } const), !(v \texttt{ == } const) \mid \forall v \forall const, \text{such that } \sigma(v) = const\}$
    $cnt \longleftarrow 0$
    while $C \neq \emptyset$ and $cnt < 20$ do
        let $c \in C$ maximizes $F(R', S', c)$
        $C \longleftarrow C / \{c\}$
        if $\mathrm{Test}(\langle p'[c/\texttt{abstc}], n' \rangle, NegT, PosT)$ then
            return $\langle p'[c/\texttt{abstc}], n' \rangle$
        $cnt \longleftarrow cnt + 1$
else
    if $\mathrm{Test}(\langle p', n' \rangle, NegT, PosT)$ then
        return $\langle p', n' \rangle$
return $\emptyset$
```

Figure 4-5: Patch Validation Algorithm with Condition Synthesis

$$\text{Exec}(\langle p, n\rangle, I, D) = \begin{cases} \langle O, R, S\rangle & \exists I\prime, O, R, S, \text{such that } \langle \ell_0, \sigma_0, I, \epsilon, D, \epsilon, \epsilon\rangle =\!\llbracket \langle p, n\rangle \rrbracket\!\Rightarrow \\ & \langle \texttt{nil}, \sigma, I\prime, O, D\prime, R, S\rangle \\ \bot & \text{otherwise} \end{cases}$$

$$\text{Test}(\langle p, n\rangle, \mathit{NegT}, \mathit{PosT}) = \begin{cases} \texttt{False} & \exists\langle I, O\rangle \in (\mathit{NegT} \cup \mathit{PosT}), \text{such that} \\ & \text{Exec}(\langle p, n\rangle, I, \epsilon) = \langle O\prime, R, S\rangle,\ O \neq O\prime \\ \texttt{True} & \mathit{otherwise} \end{cases}$$

$$F(\epsilon, \epsilon, c) = 0 \qquad \frac{\sigma \vdash c \Rightarrow x}{F(x \circ R, \sigma \circ S, c) = F(R, S, c) + 1} \qquad \frac{\sigma \vdash c \Rightarrow (1 - x)}{F(x \circ R, \sigma \circ S, c) = F(R, S, c)}$$

$$\text{Flip}(\epsilon) = \epsilon \qquad \frac{R = R\prime \circ 0}{\text{Flip}(R) = R\prime \circ 1} \qquad \frac{R = R\prime \circ 1}{\text{Flip}(R) = \text{Flip}(R\prime)}$$

Figure 4-6: Definitions of Exec, Test, Flip, and F

this case the execution is transfered to the corresponding labels with D, R, and S unchanged. The third and the fourth rules specify the case where there are no more future abstract condition values in D for the abstract condition `abstc`. These rules use the semantics-preserving value for the abstract condition `abstc`, with R and S appropriately updated. The last four rules specify the case where D is not empty. In this case the execution continues as if the abstract condition returns the next value in the sequence D, with R, and S updated accordingly.

For each negative test case, the algorithm in Figure 4-5 searches a sequence of abstract condition values with the goal of finding a sequence of values that generates the correct output for the test case (lines 4-18). Flip is an utility function that enables Prophet to explore different abstract condition value sequences (see Figure 4-6). The algorithm (line 12) executes the program with different future abstract condition value sequences D to search for a sequence that passes each negative test case. If the algorithm cannot find such a sequence for any negative test case, it will move on to the next current candidate program (line 15).

Prophet and SPR try a configurable number (in our current implementation, 11) of different abstract condition value sequences for each negative test case in the loop

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{\&\& !abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 0}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_2, \sigma, I, O, D, R, S\rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{|| abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 1}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_1, \sigma, I, O, D, R, S\rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{\&\& !abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 1}{\langle \ell, \sigma, I, O, \epsilon, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_1, \sigma, I, O, \epsilon, R \circ 0, S \circ \sigma\rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{|| abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 0}{\langle \ell, \sigma, I, O, \epsilon, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_2, \sigma, I, O, \epsilon, R \circ 0, S \circ \sigma\rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{\&\& !abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 1 \qquad D = 0 \circ D'}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_1, \sigma, I, O, D', R \circ 0, S \circ \sigma\rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{|| abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 0 \qquad D = 0 \circ D'}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_2, \sigma, I, O, D', R \circ 0, S \circ \sigma\rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{\&\& !abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 1 \qquad D = 1 \circ D'}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_2, \sigma, I, O, D', R \circ 1, S \circ \sigma\rangle}$$

$$\frac{p(\ell) = \texttt{if (}c\ \texttt{|| abstc)}\ \ell_1\ \ell_2 \qquad \sigma \vdash c \Rightarrow 0 \qquad D = 1 \circ D'}{\langle \ell, \sigma, I, O, D, R, S\rangle \mathrel{=\!\!\llbracket} \langle p, n\rangle \mathrel{\rrbracket\!\!\Rightarrow} \langle \ell_1, \sigma, I, O, D', R \circ 1, S \circ \sigma\rangle}$$

Figure 4-7: Small Step Operational Semantics for if-statements with Abstract Condition

(lines 7-13). At each iteration (except the last) of the loop, the algorithm flips the last non-zero value in the previous abstract condition value sequence (see Flip definition in Figure 4-6). In the last iteration Prophet flips all abstract condition values to 1 (line 9 in Figure 4-5) (note that the program may executes an abstract condition multiple times).

The rationale is that, in practice, if a negative test case exposes an error at an if statement, either the last few executions of the if statement or all of the executions take the wrong branch direction. This empirical property holds for all defects in our benchmark set.

If a future abstract condition value sequence can be found for every negative test case, the algorithm concatenates the found sequences $R\prime$ and the corresponding recorded environments to $S\prime$ (lines 17-18). The algorithm then executes the candidate program with the positive test cases and concatenates the sequences and the recorded environments as well (lines 21-22). Note that for positive cases the algorithm simply returns zero for all abstract conditions, so that the candidate program has the same execution as the original program.

Condition Generation: The algorithm enumerates all conditions in the search space and evaluates each condition against the recorded condition values (R$\prime$) and environments ($S\prime$). It counts the number of recorded condition values that the condition matches. Our current condition space is the set of all conditions of the form `(`v `==` $const$`)` or `!(`v `==` $const$`)` such that $\exists \sigma \in S\prime . \sigma(v) = const$. It is straightforward to extend this space to include comparison operators $(<, \leq, \geq, >)$ and a larger set of logical expressions. For our benchmark set of defects, the relatively simple Prophet and SPR condition space contains a remarkable number of correct patches, with extensions to this space delivering relatively few additional correct patches (see Section 5.5).

We define F(R$\prime$, S$\prime$, c) in Figure 4-6, which counts the number of branch directions for the condition c that match the recorded abstract condition values $R\prime$ given the recorded environments $S\prime$. The algorithm enumerates a configurable number (in our current implementation, 20) of the top conditions that maximize $F(R\prime, S\prime, c)$ (lines 26-32). The algorithm then validates the transformed candidate program with the abstract condition replaced by the generated condition c (lines 30-31). $p[c/\texttt{abstc}]$ denotes the result of replacing every occurrence of abstc in p with the condition c.

Enumerating all conditions in the space is feasible because the overwhelming majority of the candidate transformed programs will not pass the condition value search stage. Prophet will therefore perform the condition generation stage very infrequently and only when there is some evidence that transforming the target condition may actually deliver a correct patch. In our experiments, Prophet performs the condition generation stage for less than 1% of the candidate transformed programs that contain an abstract condition. (see Section 5.4.4).

Alternate Condition Synthesis Techniques: It is straightforward to implement a variety of different condition synthesis techniques. For example, it is possible to synthesize complete replacements for conditions of if statements (instead of conjoining or disjoining new conditions to existing conditions). The condition value search would start with the sequence of branch directions at that if statement with the original condition, then search for a sequence of branch directions that would generate correct outputs for all negative inputs. Condition generation would then work with the recorded branch directions to deliver a new replacement condition.

The effectiveness of condition value search in eliminating unpromising conditions enables the Prophet condition generation algorithm to simply enumerate and test all conditions in the condition search space. It is of course possible to apply arbitrarily sophisticated condition generation algorithms, for example by leveraging modern solver technology [73]. One issue is that there may be no condition that exactly matches the recorded sequences of environments and branch directions. Even if this occurs infrequently (as we would expect in practice), requiring an exact match may eliminate otherwise correct patches. An appropriate solver may therefore need to generate approximate solutions.

4.2.4 SPR Patch Validation Order

Given a program $\langle p, n \rangle$, both Prophet and SPR first apply transformation schemas described in Section 4.2.2 to obtain the set of candidate patches $M(\langle p, n \rangle)$. Prophet and SPR then use the validation algorithm in Section 4.2.3 to validate each candidate patch one by one. Prophet uses a machine learning algorithm to determine the patch validation order, which we will present in Section 4.3.

Unlike Prophet, SPR uses a set of manual heuristic rules to determine the patch validation order. SPR empirically sets the validation order as follows:

1. SPR first tests patches that change only a branch condition (e.g., tighten and loosen a condition).

2. SPR tests patches that insert an if-statement before a statement s, where s is the first statement of a compound statement (i.e., C code block).

3. SPR tests patches that insert an if-guard around a statement s.

4. SPR tests patches that insert a memory initialization.

5. SPR tests patches that insert an if-statement before a statement s, where s is not the first statement of a compound statement.

6. SPR tests patches a) that replace a statement or b) that insert a non-if statement (i.e., generated by $\mathrm{M}_{\mathbf{CopyRep}}$) before a statement s where s is the first statement of a compound statement.

7. SPR finally tests the remaining patches.

Intuitively, SPR prioritizes patches that contain conditionals. With abstract conditions that SPR later synthesizes, each condition value search stands in for multiple potential patches. SPR also prioritizes patches that insert a statement before the first statement of a compound statement (i.e., a code block), because inserting statements at other locations is often semantically equivalent to such patches.

If two patches have the same tier in the previous list, their test orders are determined by the rank of the two corresponding original statements (which two patches are based on) in the list returned by the defect localizer.

4.3 Learning Universal Properties

This section presents the learning algorithm in Prophet. The learning algorithm guides the exploration of the search space in Prophet. It operates with a probabilistic model that identifies and prioritize potentially correct patches in the search space. We first discuss how Prophet works with the patches that do not contain abstract conditions in the Prophet search space. We then extend the treatment to patches with abstract conditions (see Section 4.3.5)

4.3.1 Probabilistic Model

Given a defective program p and a search space of candidate patches, the Prophet probabilistic model is a parameterized likelihood function which assigns each candidate patch δ a probability $P(\delta \mid p, \theta)$, which indicates how likely δ is a correct patch for p. θ is the model parameter vector which Prophet learns during its offline training phase (see Section 4.3.3). Once θ is determined, the probability can be interpreted as a normalized score (i.e., $\sum_\delta P(\delta \mid p, \theta) = 1$) which prioritizes correct patches among all possible candidate patches.

The Prophet probabilistic model assumes that each candidate patch δ in the search space can be derived from the given defective program p in two steps: 1) Prophet selects a program point $\ell \in L(p)$, where $L(p)$ denotes the set of program points in p that Prophet may attempt to modify and 2) Prophet selects an AST modification operation $m \in M(p, \ell)$ and applies m at ℓ to obtain δ, where $M(p, \ell)$ denotes the set of all possible modification operations that Prophet may attempt to apply at ℓ.

Therefore the patch δ is a pair $\langle m, \ell \rangle$. We define $P(\delta \mid p, \theta) = P(m, \ell \mid p, \theta)$ for $\ell \in L(p)$ and $m \in M(p, \ell)$ as follows:

$$P(m, \ell \mid p, \theta) = \frac{1}{Z} \cdot A \cdot B$$

$$A = (1 - \beta)^{r(p, \ell)}$$

$$B = \frac{\exp(\phi(p, m, \ell) \cdot \theta)}{\sum_{\ell' \in L(p)} \sum_{m' \in M(p, \ell')} \exp(\phi(p, m', \ell') \cdot \theta)}$$

Here B is a standard parameterized log-linear distribution determined by the extracted feature vectors ϕ and the learned parameter vector θ. A is a geometric distribution that encodes the information Prophet obtains from its defect localization algorithm (which identifies target program points to patch). The algorithm performs a dynamic analysis on the execution traces of the program p on the supplied test suite to obtain a ranked set $L(p)$ of candidate program points to modify (see Section 4.4.1). $r(p, \ell)$ denotes the rank of $\ell \in L(p)$ assigned by the defect localization algorithm. Here β is the parameter of the geometric distribution (which Prophet empirically sets to 0.02).

We use a geometric distribution for the defect localization information because previous defect localization work reports that statements with higher localization ranks are significantly more likely to be patched than statements with lower localization ranks [48, 107]. The Prophet geometric geometric distribution matches this observation of previous work.

Intuitively, the formula assigns the weight $e^{\phi(p,m,\ell)\cdot\theta}$ to each candidate patch $\langle m, \ell \rangle$ based on the extracted feature vector $\phi(p, m, \ell)$ and the learned parameter vector θ. The formula then computes the weight proportion of each patch over the total weight of the entire search space derived from the functions L and M. The formula obtains the final patch probability by multiplying the weight proportion of each patch with a geometric distribution probability, which encodes the defect localization ranking of the patch.

Note that $L(p)$, $r(p, \ell)$, and $M(p, \ell)$ are inputs to the probabilistic model. $M(p, \ell)$ defines the patch search space while $L(p)$ and $r(p, \ell)$ define the defect localization algorithm. The model can work with arbitrary $L(p)$, $r(p, \ell)$, and $M(p, \ell)$, i.e., it is independent of the underlying search space and the defect localization algorithm. It is straightforward to extend the Prophet model to work with patches that modify multiple program points.

4.3.2 Defect Localization Approximation for Learning

The input to the Prophet training phase is a large set of revision changes $D = \{\langle p_1, \delta_1\rangle, \ldots, \langle p_n, \delta_n\rangle\}$, where each element of D is a pair of a defective program p_i and the corresponding successful human patch δ_i. Prophet learns a model parameter θ such that the resulting probabilistic model assigns a high conditional probability score to δ_i among all possible candidate patches in the search space.

It is, in theory, possible to learn θ directly over $P(m, \ell \mid p, \theta)$. But obtaining the defect localization information requires 1) a compiled application that runs in the Prophet training environment and 2) a test suite that includes both positive and negative test cases (and not just a standard set of regression test cases for which the unpatched application produces correct output).

The Prophet learning algorithm therefore uses an oracle-like defect localization approximation to drive the training. For each training pair $\langle p_i, \delta_i\rangle$, the algorithm computes the structural AST difference that the patch δ_i induces to 1) locate the modified program location ℓ_i and 2) identify a set of program points L'_i near ℓ_i (i.e., in the same basic block as ℓ_i and within three statements of ℓ_i in this basic block). It then uses maximum likelihood estimation to learn θ over the following formula:

$$\theta = \arg\max_{\theta} \left(\sum_i \log C_i - \lambda_1 \sum_j |\theta_j| - \lambda_2 \sum_j \theta_j^2 \right)$$

$$C_i = \frac{\exp\left(\phi(p_i, m_i, \ell_i) \cdot \theta\right)}{\sum_{\ell' \in L'_i} \sum_{m' \in M(p_i, \ell')} \exp\left(\phi(p_i, m', \ell') \cdot \theta\right)}$$

λ_1 and λ_2 are L1 and L2 regularization factors which Prophet uses to avoid overfitting. Prophet empirically sets both factors to 10^{-3}.

Using the defect localization approximation (as opposed to full defect localization) provides at least two advantages. First, it significantly expands the range of applications from which Prophet can draw training patches — it enables Prophet to work with successful human patches from applications even if the application does not fully compile and execute in the Prophet training environment and even if the application does not come with an appropriate test suite. Second, it also improves the running time of the training phase (which takes less than two hours in our experiments, see Chapter 5), because Prophet does not need to compile and run patches during training.

4.3.3 Learning Algorithm

```
Input  : the training set $D = \{\langle p_1, \delta_1 \rangle, \ldots, \langle p_n, \delta_n \rangle\}$, where $p_i$ is the original program
         and $\delta_i$ is the successful human patch for $p_i$.
Output : the feature weight parameter vector $\theta$.
for $i = 1$  to $n$ do
    $\langle m_i, \ell_i \rangle \longleftarrow \delta_i$
    $L'_i \longleftarrow \text{NearLocations}(p_i, \ell_i)$
$n_0 \longleftarrow 0.85 \cdot n$
Initialize all elements in $\theta$ to 0
$\theta^* \longleftarrow \theta$
$\alpha \longleftarrow 1$
$\gamma^* \longleftarrow 1$
$cnt \longleftarrow 0$
while $cnt < 200$ do
    Assume $g(p, \ell, m, L, \theta) = e^{\phi(p,m,\ell) \cdot \theta} / (\Sigma_{\ell' \in L} \Sigma_{m' \in M(p,\ell')} e^{\phi(p,m',\ell') \cdot \theta})$
    Assume $f(\theta) = \frac{1}{n_0} \cdot \Sigma_{i=1}^{n_0} \log g(p_i, \ell_i, m_i, L'_i, \theta) - \lambda_1 \cdot \Sigma_{i=1}^{k} |\theta_i| - \lambda_2 |\theta|^2$
    $\theta \longleftarrow \theta + \alpha \cdot \frac{\partial f}{\partial \theta}$
    $\gamma \longleftarrow 0$
    for $i = n_0 + 1$  to $n$ do
        $tot \longleftarrow |\{m \mid m \in M(p_i, \ell),\ \ell \in L'_i\}|$
        $rank \longleftarrow |\{m \mid m \in M(p_i, \ell),\ \ell \in L'_i,$
            $g(p_i, \ell, m, L'_i, \theta) \geq g(p_i, \ell_i, m_i, L'_i, \theta)\}|$
        $\gamma \longleftarrow \gamma + (rank/tot)/(n - n_0)$
    if $\gamma < \gamma^*$ then
        $\theta^* \longleftarrow \theta$
        $\gamma^* \longleftarrow \gamma$
        $cnt \longleftarrow 0$
    else
        $cnt \longleftarrow cnt + 1$
        if $\alpha > 0.01$ then
            $\alpha \longleftarrow 0.9 \cdot \alpha$
return $\theta^*$
```

Figure 4-8: Learning Algorithm

Figure 4-8 presents the Prophet learning algorithm. Combining standard machine learning techniques, Prophet computes θ via gradient descent as follows:

- **AST Structural Difference:** For each pair $\langle p_i, \delta_i \rangle$ in D, Prophet computes the AST structural difference of δ_i to obtain the corresponding modification operation m_i and the modified program point ℓ_i (lines 1-3). The function NearLocations(p_i, ℓ_i) at line 3 returns a set of program points that are close to the known correct modification point ℓ_i.

- **Initialization:** Prophet initializes θ with all zeros. Prophet also initializes the learning rate of the gradient descent (α at line 7) to one. At line 4, Prophet splits the training set and reserves 15% of the training pairs as a validation set. Prophet uses this validation set to measure the performance of the learning process and avoid overfitting. Prophet uses the remaining 85% of the training pairs to perform the gradient descent computation.

- **Update Current θ:** Prophet runs an iterative gradient descent algorithm. Prophet updates θ at lines 11-13 at the start of each iteration.

- **Measure Performance:** For each pair of $\langle p_i, \delta_i \rangle$ in the validation set, Prophet computes the percentage of candidate programs in the search space that have a higher probability score than δ_i (lines 15-18). Prophet uses the average percentage (γ) over all of the validation pairs to measure the performance of the current θ. Lower percentage is better because it indicates that the learned model ranks correct patches higher among all candidate patches.

- **Update Best θ and Termination:** θ^* in Figure 4-8 corresponds to the best observed θ. At each iteration, Prophet updates θ^* at lines 19-22 if the performance (γ) of the current θ on the validation set is better than the best previously observed performance (γ^*). Prophet decreases the learning rate α at lines 25-26 if θ^* is not updated. If it does not update θ^* for 200 iterations, the algorithm terminates and returns θ^* as the result.

4.3.4 Feature Extraction

$$\begin{array}{lll} c & := & c_1 \ \texttt{\&\&}\ c_2 \mid c_1 \ \texttt{||}\ c_2 \mid v\texttt{!=}\mathit{const} \mid v\texttt{==}\mathit{const} \\ \mathit{simps} & := & v\ \texttt{=}\ v_1\ \texttt{op}\ v_2 \mid v\ \texttt{=}\ \mathit{const} \mid \texttt{print}\ v \\ & & \mid \texttt{skip} \mid \texttt{break} \\ s & := & \ell : \mathit{simps} \mid \{\ s_1\ s_2\ \ldots\ \} \mid \ell : \texttt{if}\ (c)\ s_1\ s_2 \\ & & \mid \ell : \texttt{while}\ (c)\ s_1 \\ p & := & \{\ s_1\ s_2\ \ldots\ \} \end{array}$$

$$v,\ v_1,\ v_2 \in \mathbf{Var} \quad \mathit{const} \in \mathbf{Int} \quad \ell \in \mathbf{Label}$$
$$c,\ c_1,\ c_2 \in \mathbf{Cond} \quad s,\ s_1,\ s_2 \in \mathbf{Stmt}$$
$$p \in \mathbf{Prog} \quad \mathbf{Atom} = \mathbf{Var} \cup \mathbf{Int}$$

Figure 4-9: The Language for Illustrating Feature Extraction

$$\begin{array}{ll} \mathbf{Patch} = \mathbf{Modification} \times \mathbf{Label} & \mathbf{Pos} = \{\texttt{C}, \texttt{P}, \texttt{N}\} \\ \mathbf{MK} = \{\texttt{InsertControl}, \texttt{InsertGuard}, \texttt{ReplaceCond}, & \\ \quad \texttt{ReplaceStmt}, \texttt{InsertStmt}\} & \\ \mathbf{SK} = \{\texttt{Assign}, \texttt{Print}, \texttt{While}, \texttt{Break}, \texttt{Skip}, \texttt{If}\} & \\ \mathbf{ModFeature} = \mathbf{MK} \cup (\mathbf{Pos} \times \mathbf{SK} \times \mathbf{MK}) & \\ \mathbf{ValueFeature} = \mathbf{Pos} \times \mathbf{AC} \times \mathbf{AC} & \end{array}$$

$$\begin{array}{ll} \text{Stmt}: & \mathbf{Prog} \times \mathbf{Label} \rightarrow \mathbf{Stmt} \\ \text{ApplyPatch}: & \mathbf{Prog} \times \mathbf{Patch} \rightarrow \mathbf{Prog} \times (\mathbf{Cond} \cup \mathbf{Stmt}) \\ \text{ModKind}: & \mathbf{Modification} \rightarrow \mathbf{MK} \\ \text{StmtKind}: & \mathbf{Stmt} \rightarrow \mathbf{SK} \\ \psi: & \mathbf{Prog} \times \mathbf{Atom} \times (\mathbf{Cond} \cup \mathbf{Stmt}) \rightarrow \mathbf{AC} \\ \text{FIdx}: & (\mathbf{ModFeature} \cup \mathbf{ValueFeature}) \rightarrow \mathbf{Int} \\ & \forall a, \forall b, (\mathit{FIdx}(a) = \mathit{FIdx}(b)) \iff (a = b) \end{array}$$

Figure 4-10: Definitions and Notation. **SK** corresponds to the set of statement kinds. **MK** corresponds to the set of modification kinds. **AC** corresponds to the set of atomic characteristics that the analysis function ψ extracts.

Figure 4-9 presents the syntax of a simple programming language which we use to present the Prophet feature extraction algorithm (see the end of this section for a discussion of how we extend the feature extraction algorithm for C programs). Each of the statements (except compound statements) is associated with a unique label ℓ. A program p in the language corresponds to a compound statement. The semantics of the language in Figure 4-9 is similar to C. For brevity, we omit the operational semantics.

Figure 4-10 presents the notation we use to present the feature extraction algorithm. Figure 4-11 presents the feature extraction algorithm itself. Given a program p, a program point ℓ, and a modification operation m that is applied at ℓ, Prophet extracts features by analyzing both m and the original code near ℓ.

```
Input  : the input program $p$, the modified program point $\ell$, and the modification
         operation $m$
Output : the extracted feature vector $\phi(p, \ell, m)$
Initialize all elements in $\phi$ to 0
$S_{\mathtt{C}} \longleftarrow \{\mathrm{Stmt}(p, \ell)\}$
$S_{\mathtt{P}} \longleftarrow \mathrm{Prev3stmts}(p, \ell))$
$S_{\mathtt{N}} \longleftarrow \mathrm{Next3stmts}(p, \ell))$
$idx \longleftarrow \mathrm{FIdx}(\mathrm{ModKind}(m))$
$\phi_{idx} \longleftarrow 1$
for $i$ in $\{C, P, N\}$ do
    for $s$ in $S_i$ do
        $idx \longleftarrow \mathrm{Fid}(\langle i, \mathrm{StmtKind}(s), \mathrm{ModKind}(m)\rangle)$
        $\phi_{idx} \longleftarrow 1$
$\langle p', n\rangle \longleftarrow \mathrm{ApplyPatch}(p, \langle m, \ell\rangle)$
for $i$ in $\{C, P, N\}$ do
    for $a$ in $\mathrm{Atoms}(n)$ do
        for $s$ in $S_i$ do
            for $ac'$ in $\psi(p', a, n)$ do
                for $ac$ in $\psi(p, a, s)$ do
                    $idx \longleftarrow \mathrm{FIdx}(\langle i, ac, ac'\rangle)$
                    $\phi_{idx} \longleftarrow 1$
return $\phi$
```

Figure 4-11: Feature Extraction Algorithm

$$\psi : \mathbf{Prog} \times \mathbf{Atom} \times (\mathbf{Cond} \cup \mathbf{Stmt}) \to \mathbf{AC} \qquad \psi(p, a, \mathit{node}) = \psi_0(a, \mathit{node}) \cup \psi_1(a, \mathit{node})$$

$$\mathbf{AC} = \{\texttt{var}, \texttt{const0}, \texttt{constn0}, \texttt{cond}, \texttt{if}, \texttt{prt}, \texttt{loop}, \texttt{==}, \texttt{!=}, \langle \text{op}, \texttt{L}\rangle, \langle \text{op}, \texttt{R}\rangle, \langle \texttt{=}, \texttt{L}\rangle, \langle \texttt{=}, \texttt{R}\rangle\}$$

$$\frac{v \in \mathbf{Var}}{\psi_0(v, \mathit{node}) = \{\texttt{var}\}} \qquad \frac{\mathit{const} = 0}{\psi_0(\mathit{const}, \mathit{node}) = \{\texttt{const0}\}} \qquad \frac{\mathit{const} \in \mathbf{Int} \qquad \mathit{const} \neq 0}{\psi_0(\mathit{const}, \mathit{node}) = \{\texttt{constn0}\}}$$

$$\frac{a \notin \text{Atoms}(\mathit{node})}{\psi_1(a, \mathit{node}) = \emptyset} \qquad \frac{c = \text{“}v\texttt{==}\mathit{const}\text{”}}{\psi_1(v, c) = \{\texttt{cond}, \texttt{==}\} \qquad \psi_1(\mathit{const}, c) = \{\texttt{cond}, \texttt{==}\}}$$

$$\frac{c = \text{“}v\,\texttt{!=}\mathit{const}\text{”}}{\psi_1(v, c) = \{\texttt{cond}, \texttt{!=}\} \qquad \psi_1(\mathit{const}, c) = \{\texttt{cond}, \texttt{!=}\}}$$

$$\frac{c = \text{“}c_1 \;\texttt{\&\&}\; c_2\text{” or } c = \text{“}c_1 \;\texttt{||}\; c_2\text{”} \qquad a \in \text{Atoms}(c)}{\psi_1(a, c) = \psi_1(a, c_1) \cup \psi_1(a, c_2)}$$

$$\frac{s = \text{“}\ell : v = v_1 \text{ op } v_2\text{”}}{\psi_1(v, s) = \{\langle \texttt{=}, \texttt{L}\rangle\} \qquad \psi_1(v_1, s) = \{\langle \text{op}, \texttt{L}\rangle, \langle \texttt{=}, \texttt{R}\rangle\} \qquad \psi_1(v_2, s) = \{\langle \text{op}, \texttt{R}\rangle, \langle \texttt{=}, \texttt{R}\rangle\}}$$

$$\frac{s = \text{“}\ell : v\text{=}\mathit{const}\text{”}}{\psi_1(v, s) = \{\langle \texttt{=}, \texttt{L}\rangle\} \qquad \psi_1(\mathit{const}, s) = \{\langle \texttt{=}, \texttt{R}\rangle\}} \qquad \frac{s = \text{“}\ell : \texttt{print}\;\; v\text{”}}{\psi_1(v, s) = \{\texttt{prt}\}}$$

$$\frac{s = \text{“}\ell : \texttt{while}\;\; (c)\;\; s_1\text{”} \qquad a \in \text{Atoms}(s)}{\psi_1(a, s) = \psi_1(a, c) \cup \psi_1(a, s_1) \cup \{\texttt{loop}\}} \qquad \frac{s = \text{“}\{s_1 s_2 \ldots\}\text{”} \qquad a \in \mathit{Atoms}(s)}{\psi_1(a, s) = \psi_1(a, s_1) \cup \psi_1(a, s_2) \cup \cdots}$$

$$\frac{s = \text{“}\ell : \texttt{if}\;\; (c)\;\; s_1\;\; s_2\text{”} \qquad a \in \text{Atoms}(s)}{\psi_1(a, s) = \psi_1(a, c) \cup \psi_1(a, s_1) \cup \psi_1(v, s_2) \cup \{\texttt{if}\}}$$

Figure 4-12: Atomic Characteristic Extraction Rules for $\psi(p, a, n)$

Prophet first partitions the statements near ℓ in the original program p into three sets S_{C}, S_{P}, and S_{N} based on the relative positions of the statements (lines 1-3). S_{C} contains only the statement associated with the modification point ℓ (returned by the utility function Stmt). S_{P} contains the statements that appear at most three statements before ℓ in the enclosing compound statement (returned by the utility function Prev3stmts). S_{N} contains the statements that appear at most three statements after ℓ in the enclosing compound statement (returned by the utility function Next3stmts).

Prophet then extracts two types of features, modification features (lines 5-10) and program value features (lines 11-18). Modification features capture interactions between the modification m and the surrounding statements, while program value features capture how the modification works with program values (i.e., variables and constants) in the original and patched code. For each extracted feature, Prophet sets the corresponding bit in ϕ whose index is identified by the utility function FIdx (lines 5-6, lines 9-10, and lines 17-18). FIdx maps each individual feature to a unique integer value.

Modification Features: Prophet implements two classes of modification features. The first class captures the kind of modification that m applies. The second class captures relationships between the kinds of statements that appear near the patched statement in the original program and the modification kind of m. So, for example, if successful patches often insert a guard condition before a call statement, a modification feature will enable Prophet to recognize and exploit this fact.

At lines 5-6 in Figure 4-11, Prophet extracts the modification kind of m as the modification feature. At lines 7-10, Prophet also extracts the triple of the position of an original statement relative to the patched statement ("C" corresponds to the original statement, "P" corresponds to one of the three previous statements in the same block, and "N" corresponds to one of the three next statements in the same block), the kind of the original statement, and the modification kind of m as the modification feature. At line 9, the utility function StmtKind(s) returns the statement kind of s and the utility function ModKind(m) returns the modification kind of m.

Prophet currently classifies modification operations into five kinds: `InsertControl` (inserting a potentially guarded control statement before a program point), `AddGuardCond` (adding a guard condition to an existing statement), `ReplaceCond` (replacing a branch condition), `InsertStmt` (inserting a non-control statement before a program point), and `ReplaceStmt` (replacing an existing statement). See Figure 4-10 for the definition of modification features, statement kinds, and modification kinds.

Program Value Features: Program value features are designed to capture relationships between how variables and constants are used in the original program and how they are used in the patch. For example, if successful patches often insert a check involving a variable that is subsequently passed as a parameter to a nearby call statement, a program value feature will enable Prophet to recognize and exploit this fact. Program value features capture interactions between an occurrence of a variable or constant in the original program and an occurrence of the same variable or constant in the new code in the patch.

To avoid polluting the feature space with application-specific information, program value features abstract away the specific names of variables and values of constants involved in the interactions that these features model. This abstraction enables Prophet to learn properties of correct code as captured by program value features from patches for one set of applications, then apply the learned information to generate correct patches for other applications.

To extract program value features, Prophet first applies the patch to the original program (line 11 in Figure 4-11). $\text{ApplyPatch}(p, \langle m, \ell \rangle)$ denotes the results of the patch application, which produces a pair $\langle p', n \rangle$, where p' is the new patched program and n is the AST node for the new statement or condition that the patch introduces. Prophet performs a static analysis on both the patched and original programs to extract a set of atomic characteristics for each program atom a (i.e., a variable or an integer). In Figure 4-11, $\psi(p, a, n)$ denotes the set of atomic characteristics extracted for a in n.

At lines 12-18, Prophet extracts each program value feature, which is a triple $\langle i, ac, ac' \rangle$ of the position i of a statement in the original program, an atomic characteristic ac of a program atom in the original statement, and an atomic characteristic ac' of the same program atom in the AST node that the patch introduces. Intuitively, the program value features track co-occurrences of each pair of the atomic characteristic ac in the original code and the atomic characteristic ac' in the modification m. The utility function Atoms(n) at line 12 returns a set that contains all program atoms (i.e., program variables and constants) in n.

Figure 4-12 presents the static analysis rules that Prophet uses to extract atomic characteristics $\psi(p, v, n)$. These rules track the roles that v plays in the enclosing statements or conditions and record the operations in which v participates. The first three rules in Figure 4-12 track whether an expression atom is a variable, a zero constant, or a non-zero constant. The fourth through eleventh rules track statement types and operators that are associated with each expression atom. The last three rules recursively compute and propagate atomic characteristics for if statements, statement blocks, and while statements, respectively.

Note that Prophet can work with any static analysis to extract arbitrary atomic characteristics. It is therefore possible, for example, to combine Prophet with more sophisticated analysis algorithms to obtain a richer set of atomic characteristics.

Feature Extraction for C: Prophet extends the feature extraction algorithm to C programs as follows. Prophet treats call expressions in C as a special statement kind for feature extraction. Prophet extracts atomic characteristics for binary and unary operations in C. For each variable v, Prophet also extracts atomic characteristics that capture the scope of the variable (e.g., global or local) and the type of the variable (e.g., integer, pointer, pointer to structure). The current Prophet implementation tracks over 30 atomic characteristics (see Table 4.1 for a list of these atomic characteristics) and works with a total of 3515 features, including 455 modification features and 3060 program value features.

4.3.5 Feature Extraction for Abstract Conditions

Some of the transformation schemas in Prophet contain abstract conditions that the Prophet condition synthesis algorithm will later instantiate to obtain a final patch.

Commutative Operators	Is an operand of +, *, ==, or !=
Binary Operators	Is a left/right operand of -, /, <, >, <=, >=, . (field access), -> (member access), or [] (index)
Unary Operators	Is an operand of -, ++ (increment), -- (decrement), * (dereference), or & (address-taken)
Enclosing Statements	Occurs in an assign/loop/return/if statement Occurs in a branch condition Is a function call parameter Is the callee of a call statement
Value Traits	Is a local variable, global variable, argument, struct field, constant, non-zero constant, zero, or constant string literal Has an integer, pointer, or struct pointer type Is dereferenced
Patch Related	Is the only variable in an abstract expression Is replaced by the modification operation

Table 4.1: Atomic Characteristics of Program Values for C

Specifically, Prophet implements schemas that 1) (Tighten) tighten the condition of a target if statement (by conjoining a condition C to the if condition), 2) (Loosen) loosen the condition of a target if statement (by disjoining a condition C to the if condition), 3) (Add Guard) add a guard with a condition C to a target statement, and 4) (Insert Guarded Control Flow) insert a new guarded control flow statement (if (C) return; if (C) break; or if (C) goto l; where l is an existing label in the program and C is the condition that the guard enforces) before the target statement. Here C is an abstract condition that the condition synthesis algorithm will later instantiate.

Partially Instantiated Patches: There are two obvious approaches to combine the learning algorithm and the condition synthesis. The first is to use condition synthesis to generate fully instantiated final patches during the initial generation of the search space, then use the Prophet learned model to rank these patches for validation along with all of the other candidate patches. A downside of this approach is the time required to run the condition synthesis algorithm (which compiles and executes the application potentially multiple times) to generate fully instantiated patches (many of which will have low correctness probabilities).

The second approach is to rank the uninstantiated patch (this patch has an unconstrained abstract condition C), then instantiate the abstract condition C later during patch validation. The downside of this approach is that the correctness of the final instantiated patches will depend heavily on the variables that they access. This information, of course, is not available for patches with unconstrained abstract conditions, which inhibits the ability of Prophet to compute accurate correctness probabilities for the final fully instantiated patches that the condition synthesis algorithm will generate.

Prophet therefore uses an intermediate third approach — it generates and ranks *partially instantiated patches* that specify the variable that the synthesized condition will check, but leave the abstract condition otherwise unconstrained. This approach enables Prophet to work with patches that it can acceptably accurately rank while deferring condition synthesis until patch validation time. Because deferring condition synthesis enables Prophet to move quickly on to start validating highly ranked patches, it can significantly reduce the time Prophet requires to find correct fully instantiated patches.

Learning for Partially Instantiated Patches: We extend the Prophet feature extraction algorithm to handle partially instantiated patches. Specifically, we define atomic characteristics that identify variables that the conditions in partially instantiated patches check (see Table 4.1).

The Prophet learning algorithm works with partially instantiated patches as follows. For each patch in the training set that could have been generated by a schema with an abstract condition, it derives the corresponding partially instantiated patch. It then extracts the features for this partially instantiated patch and learns over the partially instantiated patch and its extracted features (instead of learning over the fully instantiated patch).

4.3.6 Repair Algorithm

Figure 4-13 presents the Prophet repair algorithm. Prophet generates a search space of candidate patches and uses the learned probabilistic model to prioritize potentially correct patches. Specifically, Prophet performs the following steps:

- **Generate Search Space:** At line 1, Prophet runs the defect localization algorithm (DefectLocalizer(p, T)) to return a ranked list of candidate program points to modify. At lines 2-6, Prophet then generates a search space that contains candidate patches for all of the candidate program points.

- **Rank Candidate Patch:** At lines 5-6, Prophet uses the learned θ to compute the probability score for each candidate patch. At line 7, Prophet sorts all candidate patches in the search space based on their probability score. Note that the score formula at line 5 omits the constant divisor from the formula of $P(\delta \mid p, \theta)$, because it does not affect the sort results.

- **Validate Candidate Patch:** At lines 8-12, Prophet finally tests all of the candidate patches one by one in the sorted order with the supplied test suite (i.e., T). Prophet uses the validation algorithm described in Section 4.2.3 to validate candidate patches and perform condition synthesis if necessary. Prophet outputs a list of validated candidate patches.

4.3.7 Alternative Learning Objective

Prophet uses maximum likelihood estimation to learn the model parameter θ. One alternative learning objective is to minimize the sum of hinge losses as defined by a hinge-loss function $h(p, m, l, \theta)$:

$$\begin{aligned} h(p, m, l, \theta) = {} & \max_{l' \in L(p), m' \in M(p, m')} \\ & ((\phi(p, m', l') \cdot \theta - \phi(p, m, l) \cdot \theta) + \Delta(p, \langle m, l \rangle, \langle m', l' \rangle)) \end{aligned}$$

```
Input  : the original program $p$, the test suite $T$ and the learned model parameter
         vector $\theta$
Output : emit a list of validated patches
$\langle L, r \rangle \longleftarrow$ DefectLocalizer($p$, $T$)
$Candidates \longleftarrow \emptyset$
for $\ell$ in $L$ do
    for $m$ in $M(p, \ell)$ do
        $prob\_score \longleftarrow (1-\beta)^{r(p,\ell)} \cdot e^{\phi(p,\ell,m)\cdot\theta}$
        $Candidates \longleftarrow Candidates \cup \{\langle prob\_score, m, \ell \rangle\}$
$SortedCands \longleftarrow$ SortWithFirstElement($Candidates$)
for $\langle \_, m, \ell \rangle$ in $SortedCands$ do
    $\delta \longleftarrow \langle m, \ell \rangle$
    $\delta' \longleftarrow$ validate($p, \delta, T$)
    if $\delta' \neq \emptyset$ then
        emit $\delta'$
```

Figure 4-13: Prophet Repair Algorithm

Prophet can then learn θ with the following objective function:

$$\theta = \arg\min_{\theta} \left(\sum_i h(p_i, m_i, l_i, \theta) + \lambda_1 \sum_j |\theta_j| + \lambda_2 \sum_j \theta_j^2 \right)$$

Intuitively, for each correct patch in the training set, the hinge-loss function measures the score difference between the correct patch and the incorrect patch with the highest score plus the distance between these two patches. The objective function minimizes the sum of the hinge losses over all correct patches in the training set. λ_1 and λ_2 are regularization parameters, which we empirically set to 10^{-3} (we found that 10^{-3} gives the best results in our experiments). Δ is an arbitrary distance function. In our implementation, we use the euclidean distance between two feature vectors. In the ideal case, the hinge-loss learning algorithm finds a θ such that the score of the correct patch outweighs the incorrect patch with the highest score by a significant margin given by the distance between the two patches.

Although previous work has used the hinge-loss learning algorithm to successfully predict program properties such as variable names and types [87], our experimental results show that, for our set of benchmark defects, maximum likelihood estimation outperforms using the hinge-loss objective function (see Section 5.4.2).

The reason is that the hinge-loss function considers only the score of the most highly ranked incorrect patch and not the scores of the other incorrect patches. The hinge-loss algorithm therefore (unlike maximum likelihood estimation) does not directly attempt to optimize the rank of the correct patch within the full set of candidate patches. On both the training and benchmark sets, the hinge-loss algorithm is unable to find a θ that consistently ranks the correct patch within the top few patches in the search space. In this situation maximum likelihood estimation, because it considers the scores of all of the patches, produces a θ that ranks the correct patches more highly within the search space than the θ that the hinge-loss algorithm produces. The result is that the correct patches appear earlier in the validation order with maximum likelihood estimation than with the hinge loss algorithm.

4.4 Implementation

We have implemented Prophet in C++ using the clang compiler front-end [14]. Clang contains a set of APIs for manipulating the AST tree of a parsed C program, which enables Prophet to generate a repaired source code file without dramatically changing the overall structure of the source code. Existing program repair tools [55, 82, 104] often destroy the structure of the original source by inlining all header files and renaming all local variables in their generated repaired source code. Preserving the existing source code structure helps developers understand and evaluate the repair and promotes the future maintainability of the application.

4.4.1 Defect Localization

The Prophet defect localizer recompiles the given application with additional instrumentation. It inserts a call back before each statement in the source code to record a positive counter value as the timestamp of the statement execution. Prophet then invokes the recompiled application on all test cases and produces a prioritized list that contains target statements to modify based on the recorded timestamp values. Prophet prioritizes statements that 1) are executed with more negative test cases, 2) are executed with fewer positive test cases, and 3) are executed later during executions with negative test cases.

This algorithm recompiles the given application with additional instrumentation. It inserts a call back before each statement in the source code to record a positive counter value as the timestamp of the statement execution. Prophet then invokes the recompiled application on all test cases.

For a statement s and a test case i, $t(s,i)$ is the recorded execution timestamp that corresponds to the last timestamp from an execution of the statement s when the application runs with the test case i. If the statement s is not executed at all when the application runs with the test case i, then $t(s,i) = 0$.

We use the notation NegT for the set of negative test cases that expose the defect of the program and PosT for the set of positive test cases that the original program already passes. Prophet computes three scores $a(s)$, $b(s)$, $c(s)$ for each statement s:

$$\begin{aligned}
a(s) &= \mid \{i \mid t(s,i) \neq 0,\ i \in \mathrm{NegT}\} \mid \\
b(s) &= \mid \{i \mid t(s,i) = 0,\ i \in \mathrm{PosT}\} \mid \\
c(s) &= \Sigma_{i \in \mathrm{NegT}} t(s,i)
\end{aligned}$$

A statement s_1 has higher priority than a statement s_2 if $\mathrm{prior}(s_1, s_2) = 1$, where prior is defined as:

$$prior(s_1, s_2) = \begin{cases} 1 & a(s_1) > a(s_2) \\ 1 & a(s_1) = a(s_2), b(s_1) > b(s_2) \\ 1 & \begin{array}{l} a(s_1) = a(s_2), b(s_1) = b(s_2), \\ c(s_1) > c(s_2) \end{array} \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, Prophet prioritizes statements that 1) are executed with more negative test cases, 2) are executed with fewer positive test cases, and 3) are executed later during executions with negative test cases. Prophet considers the first 200 statements as potential statements for modification.

The probabilistic model and the repair algorithm are independent from the defect localization component. Prophet can integrate with any defect localization technique that returns a ranked list of target program points to patch. It is therefore possible to combine Prophet with other defect localization techniques [29, 48, 107].

4.4.2 C Program Support and Optimizations

Prophet extends the algorithm in Section 4.2 and Section 4.3 to support C programs. Prophet applies the transformation function separately to each function in a C program. When Prophet performs variable replacement or condition synthesis, it considers all variables (including local, global, and heap variables) that appear in the current transformed function. During condition generation, Prophet also searches existing conditions *c* that occur in the same enclosing compound statement (in addition to conditions of the form (*v* == *const*) and !(*v* == *const*) described above in Section 4.2.3).

When Prophet inserts control statements, Prophet considers break, return, and goto statements. When inserting return statements, Prophet generates a repair to return each constant value in the returned type that appeared in the enclosing function. When inserting goto statements, Prophet generates a repair to jump to each already defined label in the enclosing function. When Prophet inserts initialization statements, Prophet considers to call memset() to initialize memory blocks. When Prophet copies statements for C programs, Prophet considers to copy compound statements in addition to simple statements, as long as the copied code can fit into the new context.

To represent a abstract condition, Prophet inserts a function call abstract_code() into the modified condition. When Prophet tests a candidate transformed program with abstract conditions, Prophet links the program with its runtime library, which contains an implementation of abstract_cond(). The abstract_cond() in the library implements the semantics specified in Figure 4-7.

Batched Compilation: When Prophet tests candidate repairs, compilations of the repaired application may become the performance bottleneck for Prophet. To reduce the time cost of compilations, Prophet merges similar candidate repairs into a single combined repair with a branch statement. A global integer environment variable controls the branch statement, so that the batched repair will be equivalent to each individual candidate repair, when the environment variable takes a corresponding constant value. Prophet therefore only needs to recompile the merged repair once to test each of the individual candidate repairs.

Test Case Evaluation Order: Prophet always first tests each candidate repair with the negative test cases. Empirically, negative test cases tend to eliminate invalid repairs more effectively than positive test cases. Furthermore, whenever a positive test case eliminates a candidate repair, Prophet will record this positive test case and prioritize this test case for the future candidate repair evaluation.

Repairs for Code Duplicates: Programs often contain duplicate or similar source code, often caused, for example, by the use of macros or code copy/paste during application development. Prophet detects such duplicates in the source code. When Prophet generates repairs that modify one of the duplicates, it also generates additional repairs that propagate the modification to the other duplicates.

Chapter 5

Evaluation of Prophet Patch Generation System

This chapter evaluates Prophet on a benchmark set of 69 real world defects from eight large open source applications. All of our experimental results and a replication package are also available at [11]. Our evaluation consists of the following parts:

Patch Generation: We evaluate the patch generation capability of Prophet on the benchmark set and compare it with three previous patch generation systems GenProg, AE, and Kali. For each system, we report the number of defects for which the system generates correct patches or plausible patches.

Design Decisions: We evaluate the effectiveness of the Prophet learning algorithm and the impact of this algorithm on the patch generation results. We also evaluate the impact of feature selection, the learning objective, and the condition synthesis.

Search Space and Extensions: We systematically evaluate the search space design in Prophet. We consider three different search space extensions and 16 different search space configurations. We evaluate the patch generation process on each of the different search space configurations.

Sources. The previous version of this research presented in this chapter appeared in [59], [58], and [60].

Application	LoC	Tests	Defects
libtiff	77k	78	8
lighttpd	62k	295	7
php	1046k	8471	31
gmp	145k	146	2
gzip	491k	12	4
python	407k	35	9
wireshark	2814k	63	6
fbc	97k	773	2
Total			69

Table 5.1: Benchmarks for Prophet Evaluation

5.1 Benchmarks

We evaluate Prophet on 69 real world defects in eight large open source applications: libtiff, lighttpd, php, gmp, gzip, python, wireshark, and fbc. libtiff is an image processing library for the TIFF image format. lighttpd is a popular lightweight HTTP server. php is the official implementation of the interpreter for PHP programs. gmp is a widely used free library for high precision arithmetic computations on integers and floating-point numbers. gzip is a popular compression tool in Linux. python is the official implementation of the interpreter for Python programs. wireshark is a popular network package analysis application. fbc is a compile for Free Basic programs.

Table 5.1 summarizes our benchmark defects. The first column (Application) presents the name of each application. The second column (LoC) presents the number of lines of code in the application. The third column (Tests) presents the number of the test cases in the supplied test suite of the application. php is the outlier, with an order of magnitude more test cases than any other application. The fourth column (Defects) presents the number of defects in the benchmark set for each application.

This is the same benchmark set used to evaluate Kali (see Section 2.8), GenProg [55], and AE [104]. For each defect, the benchmark set contains a test suite with positive test cases for which the unpatched program produces correct outputs and at least one negative test case for which the unpatched program produces incorrect output (i.e., the negative test case exposes the defect).

This benchmark set is, to the best of our knowledge, currently the most comprehensive publicly available C data set suitable for evaluating generate-and-validate systems — other benchmark sets have fewer defects, much smaller programs, or do not have the positive and negative test cases and build infrastructures required for generate-and-validate patch generation.

Note that the original version of benchmark set is reported to contain 105 defects [55]. Our examination of the revision changes and corresponding check in entries indicates that 36 of these reported defects are not, in fact, defects. They are instead deliberate functionality changes (see Chapter 2). Because there is no defect to correct, they are therefore outside the scope of patch generation systems. We therefore exclude these functionality changes from our experiments.

5.2 Human Patch Collection and Training

This section presents how we collect successful human patches from open source repositories and how we use the collected training patches to train Prophet.

5.2.1 Collect Successful Human Patches

We used the advanced search functionality in GitHub [16] to obtain a list of open source C project repositories that 1) were started before January 1, 2010 and 2) had more than 2000 revisions. We browsed the projects from the list one by one and collected the first eight projects that 1) are command-line applications or libraries that run on our experimental environment Ubuntu 14.04, 2) whose repositories contain more than ten revision changes with patches that are within the Prophet search space, and 3) whose compilation flags can be extracted by our scripts for clang to obtain abstract syntax trees for these patches.

In this process, we considered but rejected many applications because they do not satisfy the above requirements. For example, we rejected lighttpd because it contained fewer than ten revision changes with patches within the Prophet search space. We rejected git because we were unable to extract its compilation flags using our scripts.

Project	**Revisions Used for Training**
apr	12
curl	53
httpd	75
libtiff	11
php	187
python	114
subversion	240
wireshark	85
Total	777

Table 5.2: Statistics of Collected Successful Human Patches

We stopped collecting projects when we believed we had obtained a sufficiently large training set of successful patches.

For each of the resulting eight application repositories, we ran a script to analyze the check-in logs to identify and collect all of those patches that repair defects (as opposed to changing or adding functionality) and are within the Prophet search space. From the eight repositories, we collected a total of 777 such patches. Table 5.2 presents statistics for these 777 patches.

5.2.2 Train Prophet on Collected Training Set

We train Prophet on the collected set of successful human patches. The collected set of training applications and the benchmark set share four common applications, specifically libtiff, PHP, python, and wireshark. For each of these four applications, we train Prophet separately and exclude the collected human patches of the same application from the training set. The goal is to ensure that we evaluate the ability of Prophet to apply the learned model trained with one set of applications to successfully repair defects in other applications.

The offline training takes less than two hours. Training is significantly faster than repair because the learning algorithm does not compile and run the patches in the training set during training (see Section 4.3.3). This approach enables Prophet to include patches from applications 1) for which an appropriate test suite may not be immediately available and/or 2) with relevant source code files that may not fully compile on the training platform. These two properties can significantly expand the range of applications that can contribute patches to the training set of successful human patches.

5.3 Patch Generation Evaluation

This section presents the patch generation results of Prophet and compares the results with four other patch generation systems, SPR (see Section 4.2.4), Kali (see Section 2.8), GenProg [55], and AE [104], which are evaluated on the same benchmark set.

5.3.1 Methodology

Experimental Setup: For each defect, we run the trained Prophet to obtain a sequence of validated plausible patches for that defect. For comparison, we also run SPR on each defect. We obtain the results of Kali, GenProg [55], and AE [104] on this benchmark set from Chapter 2. We terminate the execution of Prophet or SPR after 12 hours. According to the original papers, the GenProg results for each defect are obtained from 10 runs of 12 hours with different random seeds [55], the AE results for each defect are obtained from a single run of 60 hours [104].

We note that GenProg and AE require the user to specify the source file name to modify when the user applies GenProg and AE to an application that contains multiple source files (all applications in the benchmark set contain multiple source files) [85]. Prophet, SPR, and Kali do not have this limitation.

Running Environment: We run all of our experiments except those of fbc on Amazon EC2 Intel Xeon 2.6GHz machines running Ubuntu-64bit server 14.04. fbc runs only in 32-bit environments, so we run all fbc experiments on EC2 Intel Xeon 2.4GHz machines running Ubuntu-32bit 14.04.

Running Time: Although we set 12 hours as the time limit for generating patches, for the 39 defects for which Prophet finds at least one plausible patch, Prophet requires, on average, only 108.9 minutes to find and validate the first plausible patch. For the

App	LoC	Tests	Defects	Prophet	SPR	Kali	GenProg	AE
libtiff	77k	78	8	2,2	1,1	0	0	0
lighttpd	62k	295	7	0,0	0,0	0	0	0
php	1046k	8471	31	13,10	10,9	2	1	2
gmp	145k	146	2	1,1	1,1	0	0	0
gzip	491k	12	4	1,1	1,0	0	0	0
python	407k	35	9	0,0	0,0	0	0	0
wireshark	2814k	63	6	0,0	0,0	0	0	0
fbc	97k	773	2	1,1	1,0	0	0	0
Total			69	18,15	16,11	2	1	2

Table 5.3: The Number of Correct Patches Per Application Per System

15 defects for which the first validated patch is correct, Prophet requires, on average, 138.5 minutes to find and validate the first correct patch.

Evaluate Generated Patches: We manually analyze each generated plausible patch to determine whether the patch is a correct patch or just a plausible but incorrect patch that happens to produce correct outputs for all of the inputs in the test suite.

We acknowledge that, in general, determining whether a specific patch corrects a specific defect can be difficult (or in some cases not even well defined). We emphasize that this is not the case for the patches and defects that we consider here. The correct behavior for all of the evaluated defects is clear, as is patch correctness and incorrectness. Furthermore, subsequent developer patches are available for all of the defects in the benchmark set. A manual code analysis indicates that each of the correct patches in the experiments is semantically equivalent to the subsequent developer patch for that defect.

5.3.2 Experimental Results

Table 5.3 and Table 5.4 summarize the patch generation results for Prophet, SPR, Kali, GenProg, and AE. See Appendix A for the detailed experimental results of each benchmark defect. There is a row in the tables for each benchmark application. The first column (App) presents the name of the application, the second column (LoC) presents the number of lines of code in each application, and the third column (Tests)

App	LoC	Tests	Defects	Prophet	SPR	Kali	GenProg	AE
libtiff	77k	78	8	5	5	5	3	5
lighttpd	62k	295	7	3	3	4	4	3
php	1046k	8471	31	17	16	8	5	7
gmp	145k	146	2	2	2	1	1	1
gzip	491k	12	4	2	2	1	1	2
python	407k	35	9	5	5	1	0	2
wireshark	2814k	63	6	4	4	4	1	4
fbc	97k	773	2	1	1	1	1	1
Total			69	39	38	25	16	25

Table 5.4: The Number of Plausible Patches Per Application Per System

presents the number of test cases in the test suite for that application. The fourth column (Defects) presents the number of benchmark defects in each application.

The fifth through ninth columns in Table 5.3 summarize the correct patches that each system finds. For Prophet and SPR, each entry is of the form X,Y. Here X is the number of the defects for which Prophet or SPR finds a correct patch before the 12 hour timeout (even if that patch is not the first patch to validate), and Y is the number of defects for which Prophet or SPR finds a correct patch as the first patch to validate. For Kali, GenProg, and AE, each entry presents the number of the defects for which the system finds a correct patch.

The fifth through ninth columns in Table 5.4 summarize the plausible patches that each system finds. Each entry presents the number of the defects for which the corresponding system finds at least one plausible patch (i.e., a patch that passes the supplied test suite)

The results show that Prophet finds a correct patch for significantly more cases than GenProg and AE (17 more than GenProg and 15 more than AE). The results also show that Prophet finds a correct patch as the first to validate for more defects than SPR, Kali, GenProg, and AE (4 more defects than SPR, 14 more than GenProg, and 13 more than Kali and AE). One potential explanation for the underperformance of Kali, GenProg, and AE is that the correct Prophet and SPR patches are outside the search spaces of these systems (see Section 5.5), which suggests that these systems will *never* find correct patches for the remaining defects.

Both Prophet and SPR find a correct patch as the first to validate significantly more often for PHP than for other applications. There are significantly more defects for PHP than for other applications. PHP also has a much stronger test suite (an order of magnitude more test cases) than other applications, which helps both Prophet and SPR find correct patches as the first to validate.

5.3.3 Case Studies

We next present case studies of five representative benchmark defects. On the first four defects, Prophet outperforms SPR. These results illustrate the advantage of the learned patch prioritization order in Prophet over the manually defined order in SPR. The remaining defects are representative cases where Prophet does not generate correct patches or Prophet does not successfully prioritize the correct patch as the first generated patch.

php-308262-308315: Figure 5-1 presents the first patch generated by Prophet for php-308262-308315. This is a correct patch and it is identical to the developer patch for this defect. The correct patch inserts an if statement guard for an existing statement at line 11 in Figure 5-1. The inserted guard checks a function parameter variable in the existing statement against a constant value. The machine learning algorithm in Prophet detects that such relationships often correlate with successful human patches and therefore ranks this correct patch as one of top 2% in the patch validation order. Because the supplied test suite is strong enough to filter out other patches that are ranked before this correct patch, Prophet successfully generates this correct patch as the first generated patch. The SPR hand-coded heuristics rank patches that add a guard statement below patches that change a branch condition. Because of this lower rank, SPR is unable to find the correct patch within 12 hours.

gzip-a1d3d4-f17cbd: Figure 5-2 presents the first patch generated by Prophet for for gzip-a1d3d4-f17cbd. The Prophet patch is correct and semantically equivalent to the developer patch. Figure 5-3 presents the first patch generated by SPR for gzip-a1d3d4-f17cbd. The SPR patch is a plausible but incorrect patch. For gzip-a1d3d4-f17cbd, an initialization statement can be inserted at multiple candidate

```
1     static void zend_fetch_dimension_address_read(
2        temp_variable *result, zval **container_ptr,
3        zval *dim, int dim_type, int type) {
4      zval *container = *container_ptr;
5
6      switch (Z_TYPE_P(container)) {
7        ...
8        case IS_STRING: {
9          ...
10         if (Z_LVAL_P(dim) < 0 || Z_STRLEN_P(container) <= Z_LVAL_P(dim)) {
11  +        if (!(type == 3))
12             zend_error((1 << 3L), "Uninitialized string offset: %ld",
13                        (*dim).value.lval);
14           Z_STRVAL_P(ptr) = STR_EMPTY_ALLOC();
15           Z_STRLEN_P(ptr) = 0;
16         } else {
17           Z_STRVAL_P(ptr) = (char*)emalloc(2);
18           Z_STRVAL_P(ptr)[0] = Z_STRVAL_P(container)[Z_LVAL_P(dim)];
19           Z_STRVAL_P(ptr)[1] = 0;
20           Z_STRLEN_P(ptr) = 1;
21         }
22         AI_SET_PTR(result, ptr);
23         return;
24       }
25       break;
26       ...
27     }
28   }
```

Figure 5-1: Prophet Patch for php-308262-308315. Add Line 11.

```
1      void treat_stdin() {
2        ...
3    +   ifd = 0;
4        if (decompress) {
5          method = get_method(ifd);
6          if (method < 0) {
7            do_exit(exit_code);
8          }
9        }
10       if (list) {
11         do_list(ifd, method);
12         return;
13       }
14       ...
15     }
```

Figure 5-2: Prophet Patch for gzip-a1d3d4-f17cbd. Add Line 3.

```
 1    void treat_file(char *iname) {
 2  +   ifd = 0;
 3      /* Accept "-" as synonym for stdin */
 4      if (strequ(iname, "-")) {
 5        int cflag = to_stdout;
 6        treat_stdin();
 7        to_stdout = cflag;
 8        return;
 9      }
10      ...
11    }
```

Figure 5-3: SPR Patch for gzip-a1d3d4-f17cbd. Add Line 2.

```
 1    if ((td->td_nstrips > 1 && td->td_compression == 1
 2          && td->td_stripbytecount[0] != td->td_stripbytecount[1])
 3  +       && !(td->td_nstrips == 2)
 4          ) {
 5      TIFFWarning(module,
 6        "%s: Wrong \"%s\" field, ignoring and calculating from imagelength",
 7        tif->tif_name, TIFFFieldWithTag(tif, 279)->field_name);
 8      if (EstimateStripByteCounts(tif, dir, dircount) < 0)
 9        goto bad;
10    }
```

Figure 5-4: Prophet Patch for libtiff-d13be-ccadf. Add Line 3.

locations to pass the supplied test case, but not all of the resulting patches are correct. Prophet successfully prioritizes the correct patch among multiple plausible patches, because Prophet identifies many potentially useful interactions between the inserted assignment and the surrounding code via the variable "ifd". The SPR heuristics prioritize an incorrect patch that inserts the initialization at the start of a basic block.

libtiff-d13be-ccadf: Figure 5-4 presents the first patch generated by Prophet for libtiff-d13be-ccadf. The Prophet patch is correct and semantically equivalent to the developer patch. Figure 5-5 presents the first patch generated by SPR for libtiff-d13be-ccadf. The SPR patch is a plausible but incorrect patch. For libtiff-d13be-ccadf, there are multiple candidate program variables that can be used to tighten a branch condition to enable the resulting patched program to pass the supplied test suite. The learned program value features enable Prophet to successfully identify and prioritize correct patches that manipulate the right variables, which are later used in the surrounding statements following the patches. The SPR heuristics treat many

```
 1    if ((td->td_nstrips > 1 && td->td_compression == 1
 2          && td->td_stripbytecount[0] != td->td_stripbytecount[1])
 3  +       && !(1)
 4          ) {
 5      TIFFWarning(module,
 6        "%s: Wrong \"%s\" field, ignoring and calculating from imagelength",
 7        tif->tif_name, TIFFFieldWithTag(tif, 279)->field_name);
 8      if (EstimateStripByteCounts(tif, dir, dircount) < 0)
 9        goto bad;
10    }
```

Figure 5-5: SPR Patch for libtiff-d13be-ccadf. Add Line 3.

candidate clauses with the same priority and therefore generate a patch that simply eliminate the branch and disables the error checking.

fbc-5458-5459: Figure 5-6 presents the first patch generated by Prophet for fbc-5458-5459. The Prophet patch is correct and semantically equivalent to the developer patch. Figure 5-7 presents the first patch generated by SPR for fbc-5458-5459. The SPR patch is a plausible but incorrect patch. Both of the Prophet and SPR insert a clause to check the corner case where the value of the passed variable "len" is zero. The learned program value features enable Prophet to prioritize the patch in Figure 5-6 because there are many potentially useful relationships between the patch and surrounding code via the program value "len" – "len" is used multiple times in the branch at lines 21-26. SPR, on the other hand, modifies a wrong branch statement and therefore generates an incorrect patch, although passing the supplied test cases.

php-310011-310050: Figure 5-8 presents a plausible but incorrect patch generated by Prophet for php-310011-310050. This patch is ranked ahead of the generated correct patch. Figure 5-9 presents the developer patch for php-310011-310050. The code at line 4 in Figure 5-8 and Figure 5-9 sets the reference counter of the variable "tmp" to zero to free the underlying data structure. If the "tmp" variable shares the data structure with other variables, it may cause a dangling pointer error. The developer patch inserts a call to zval_copy_ctor() to separate the underlying data structure if shared. The Prophet patch however replaces line 4 with another function call to avoid setting the reference counter of "tmp" to zero. This fixes the dangling

```
 1     if((dst == NULL) || (dst->data == NULL) || (FB_STRSIZE( dst ) == 0) ) {
 2       fb_hStrDelTemp_NoLock( src );
 3       fb_hStrDelTemp_NoLock( dst );
 4       FB_STRUNLOCK();
 5       return;
 6     }
 7
 8     if((src == NULL) || (src->data == NULL) || (FB_STRSIZE( src ) == 0) ) {
 9       fb_hStrDelTemp_NoLock( src );
10       fb_hStrDelTemp_NoLock( dst );
11       FB_STRUNLOCK();
12       return ;
13     }
14
15     src_len = FB_STRSIZE( src );
16     dst_len = FB_STRSIZE( dst );
17
18     if (((start > 0) && (start <= dst_len))
19   +       && !(len == 0)
20         ) {
21       --start;
22       if ((len < 1) || (len > src_len))
23         len = src_len;
24       if (start + len > dst_len)
25         len = (dst_len - start);
26       memcpy(dst->data + start, src->data, len);
27     }
28     ...
```

Figure 5-6: Prophet Patch for fbc-5458-5459. Add Line 19.

```
1      if((dst == NULL) || (dst->data == NULL) || (FB_STRSIZE( dst ) == 0)
2   +    || (len == 0)
3        ) {
4        fb_hStrDelTemp_NoLock( src );
5        fb_hStrDelTemp_NoLock( dst );
6        FB_STRUNLOCK();
7        return;
8      }
9
10     if((src == NULL) || (src->data == NULL) || (FB_STRSIZE( src ) == 0) ) {
11       fb_hStrDelTemp_NoLock( src );
12       fb_hStrDelTemp_NoLock( dst );
13       FB_STRUNLOCK();
14       return ;
15     }
16
17     src_len = FB_STRSIZE( src );
18     dst_len = FB_STRSIZE( dst );
19
20     if (((start > 0) && (start <= dst_len))) {
21       --start;
22       if ((len < 1) || (len > src_len))
23         len = src_len;
24       if (start + len > dst_len)
25         len = (dst_len - start);
26       memcpy(dst->data + start, src->data, len);
27     }
28     ...
```

Figure 5-7: SPR Patch for fbc-5458-5459. Add Line 10.

```
1       if (Z_ISREF_PP(p)) {
2         ALLOC_INIT_ZVAL(tmp);
3         ZVAL_COPY_VALUE(tmp, *p);
4   -     Z_SET_REFCOUNT_P(tmp, 0);
5   +     zend_ptr_stack_n_pop(tmp, 0);
6         Z_UNSET_ISREF_P(tmp);
7       } else {
8         tmp = *p;
9       }
```

Figure 5-8: A Plausible But Incorrect Patch Generated by Prophet for php-310011-310050. Modify Lines 4-5.

```
1     if (Z_ISREF_PP(p)) {
2       ALLOC_INIT_ZVAL(tmp);
3       ZVAL_COPY_VALUE(tmp, *p);
4   +   zval_copy_ctor(tmp);
5       Z_SET_REFCOUNT_P(tmp, 0);
6       Z_UNSET_ISREF_P(tmp);
7     } else {
8       tmp = *p;
9     }
```

Figure 5-9: Developer Patch for php-310011-310050. Add Line 4.

```
1   - htmlNodeDumpFormatOutput(buf, docp, node, 0, format);
2   - mem = (xmlChar*) xmlBufferContent(buf);
3   - if (!mem) {
4   -   RETVAL_FALSE;
5   +   size = htmlNodeDump(buf, docp, node);
6   +   if (size >= 0) {
7   +     mem = (xmlChar*) xmlBufferContent(buf);
8   +     if (!mem) {
9   +       RETVAL_FALSE;
10  +     } else {
11  +       RETVAL_STRINGL((const char*) mem, size, 1);
12  +     }
13      } else {
14  -     RETVAL_STRING(mem, 1);
15  +     php_error_docref(NULL TSRMLS_CC, E_WARNING, "Error dumping HTML node");
16  +     RETVAL_FALSE;
17      }
18    }
```

Figure 5-10: Developer Patch for php-307563-307571. Modify Lines 1-16.

pointer error but introduces a potential memory leak error which is not tested by the supplied test suite.

This case shows that if the test suite is too weak, the learned model alone may not be able to prioritize correct patches ahead of plausible but incorrect patches. For this error, the learned model does not distinguish different function calls that manipulate the reference counter of "tmp". The model therefore does not prioritize the correct patch in Figure 5-9 ahead of the incorrect patch in Figure 5-8.

php-307563-307571: Figure 5-10 presents the developer patch for php-307563-307571. The code in Figure 5-10 converts an internal php data structure into a text string with html format. The original code invokes an incorrect function (i.e., htmlNodeDumpFormatOutput()) to implement this functionality. The patch rewrites the code block completely to properly invoke the correct function htmlNodeDump() instead.

This correct patch is outside the Prophet search space and, despite the fact that Prophet prioritizes the patch search process using information learned from successful human patches, is likely to remain beyond the reach of Prophet or similar generate and validate systems. In our experience, patch search spaces that include patches generated by transforming three or more statements can easily become intractable to search. Augmenting the search space to include such patches, in the absence of other techniques designed to improve the tractability of the search space, can leave the generate and validate system unable to find the correct patch even if the search space contains the correct patch.

5.4 Design Decision Evaluation

This section evaluates the effectiveness of the Prophet learning algorithm and the impact of it on the patch generation results. This section also evaluates several design decisions in Prophet including the feature extraction, the learning objective, and the condition synthesis.

5.4.1 Methodology

Different Variants of Prophet: To better understand how the probabilistic model, the learning algorithm, and the features affect the result, we also run five variants of Prophet, specifically Random (a naive random search algorithm that prioritizes the generated patches in a random order), Baseline (a baseline algorithm that prioritizes patches in the defect localization order, with patches that modify the same statement prioritized in an arbitrary order), MF (an variant of Prophet with only modification features; program value features are disabled), PF (a variant of Prophet with only program value features; modification features are disabled), and HL (a variant of Prophet that replaces the maximum likelihood learning with the hinge-loss learning

System	Corrected Defects	Mean Rank in Search Space
Prophet	18,15	Top 11.5%
Random	14,7	Top 41.8%
Baseline	15,8	Top 20.7%
MF	18,10	Top 12.2%
PF	18,13	Top 12.3%
HL	17,13	Top 17.0%
SPR	16,11	Top 17.5%

Table 5.5: Comparative Results for Different Systems

as described in Section 4.3.7). All of these variants, Prophet, and SPR differ only in the patch validation order, i.e., they operate with the same patch search space and the same set of optimizations for validating candidate patches.

Result Comparison between Prophet and SPR: To better understand the advantage of the learned model over the heuristic patch ranking rules in SPR, we compare the per-defect patch generation results of Prophet and SPR.

Condition Synthesis Evaluation: To better understand the advantage of the condition synthesis technique, for each Prophet run, we record the number of evaluated partially instantiated templates and the number of plausible templates (which Prophet attempts to generate concrete conditions). The difference of these two numbers indicates the number of templates that are pruned away by the Prophet validation algorithm with the condition synthesis.

5.4.2 Comparison of Different Variant Systems

Table 5.5 presents results from different patch generation systems. The first column (System) presents the name of each system (Random, Baseline, MF, and PF are variants of Prophet with different capabilities disabled, see Section 5.4.1). The second column (Corrected Defects) presents the number of the 69 defects for which the system finds at least one correct patch. Each entry is of the form X,Y, where X is the number of defects for which the system finds correct patches (whether this correct patch is the first to validate or not) and Y is the number of defects for which the system finds a correct patch as the first patch to validate.

The third column (Mean Rank in Search Space) presents a percentage number, which corresponds to the mean rank, normalized to the size of the search space for each defect, of the first correct patch in the patch prioritization order of each system. This number is an average over the 19 defects for which the search space of these systems contains at least one correct patch. We compute the size of the search space as the sum of 1) the number of partially instantiated patches generated by transformation schemas with abstract conditions and 2) the number of patches generated by other transformation schemas (these patches do not include abstract conditions). "Top X%" in an entry indicates that the corresponding system prioritizes the first correct patch as one of the top X% of the patches in the search space on average. We run the random search algorithm with the default random seed to obtain the results in the figure. The results are generally consistent with the hypothesis that the more highly a system ranks the first correct patch, the more correct patches it finds and the more correct patches it finds as the first patch to validate.

The results show that Prophet delivers the highest average rank (11.7%) for the first correct patch in the search space. The results also highlight how the Prophet model enables Prophet to successfully prioritize correct patches over plausible but incorrect patches — the Random and Baseline systems, which operate without a probabilistic model or heuristics, find a correct patch as the first to validate only roughly half as often as Prophet.

The results also show that the maximum likelihood learning algorithm in Prophet outperforms the alternative hinge-loss learning algorithm in Section 4.3.7. One potential explanation is that the hinge-loss objective function only considers two patches: the correct patch and the incorrect patch with the highest score. The maximum likelihood objective function, in contrast, considers all of the patches. The result is that the hinge-loss-trained model does not prioritize correct patches as highly in the search space as the maximum likelihood model (also see Section 4.3.7).

The results also highlight how program value features are more important than modification features for distinguishing correct patches from plausible but incorrect patches. We observed a common scenario that the search space contains multiple plausible patches that operate on different program variables. In these scenarios, the learned model with program value features enables PF (and Prophet) to identify the correct patch among these multiple plausible patches.

5.4.3 Per-Defect Results of Prophet and SPR

Table 5.6 presents the per-defect patch generation results of Prophet for the 19 benchmark defects for which the correct patch is inside the Prophet and SPR search space. Table 5.7 presents the per-defect patch generation results of SPR for the same 19 benchmark defects.

The first column (Defect) of the table presents the defect id. The second column (Search Space Templates – All) of the table presents the total number of candidate patch templates in the search space. The third column (Search Space Templates – Cond.) presents the number of patch templates that manipulate branch conditions. The fourth column (Evaluated Templates – All) presents the total number of evaluated patch templates in 12 hours. The fifth column (Evaluated Templates – Cond.) presents the total number of evaluated condition patch templates during in 12 hours. The sixth column (Plausible Templates – All) presents the number of templates for which generate plausible patches. The seventh column (Plausible Templates – Cond.)presents the number of condition templates for which generate plausible patches. The eighth column (Plausible Patches – All) presents the total number of plausible patches the system finds in 12 hours. The ninth column (Plausible Patches – Cond.) presents the number of plausible patches which manipulate branch conditions.

The tenth column (Correct Patch Gen.) presents the correct patch generation status. The number in each entry presents the number of correct patches found in 12 hours for the corresponding defect. "✓" in an entry indicates that the corresponding system successfully finds this correct patch as the first plausible patch. "△" in an entry indicates that the algorithm finds a plausible but incorrect patch as the first

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patch Gen.	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
libtiff-5b0217-3dfb33	221134	65017	64363	31834	247	149	250	152	0 X	50770	-	-
libtiff-d13be7-ccadf4	296426	228659	70050	65731	1423	1423	1703	1423	1 ✓	1183	1	1
gmp-13420-13421	50672	8763	50672	8763	3	0	3	0	2 ✓	14102	1	1
gzip-a1d3d4-f17cbd	48702	15104	48702	15104	14	0	14	0	1✓	1929	1	1
fbc-5458-5459	9857	4495	9857	4495	37	37	61	46	2 ✓	33	1	1
libtiff-ee2ce5-b5691a	171379	106867	137262	103332	328	328	328	328	1 ✓	280	1	1
php-310991-310999	89230	18988	21250	11637	1	1	1	1	1 ✓	907	1	1
php-308734-308761	14692	4160	14692	4160	4	4	4	4	2 ✓	5376	1	1
php-308262-308315	90431	10845	8496	7508	3	3	5	5	1 ✓	1365	1	1
php-307562-307561	31597	6997	14698	6997	1	0	1	0	1 ✓	2672	1	1
php-309579-309580	60351	11416	23605	11416	2	2	2	2	1 ✓	767	1	1
php-310011-310050	77671	16558	4857	4556	63	13	69	23	1 △	1348	13	21
php-309688-309716	71633	15744	3310	3241	68	68	92	92	1 △	3465	61	83
php-309516-309535	27098	6314	27098	6314	1	0	1	0	1 ✓	10954	1	1
php-307846-307853	22131	4757	16871	4709	1	0	1	0	1 ✓	10742	1	1
php-311346-311348	9799	3879	9799	3879	50	38	72	60	2 ✓	27	1	1
php-307914-307915	47988	15066	35684	15066	1	0	1	0	1 ✓	1	1	1
php-309111-309159	52908	12232	36533	11928	10	1	10	1	1 △	7701	9	9
php-309892-309910	40758	9999	13614	7118	21	17	26	22	4 ✓	462	1	1

Table 5.6: Prophet Per-defect Patch Generation Results

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patch Gen.	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
libtiff-5b0217-3dfb33	221134	65017	107534	65017	237	149	240	152	1 △	56644	208	211
libtiff-d13be7-ccadf4	296426	228659	230917	214697	1723	1723	2003	1723	1 △	372	3	3
gmp-13420-13421	50672	8763	50672	8763	3	0	3	0	2 ✓	14645	1	1
gzip-a1d3d4-f17cbd	48702	15104	48702	15104	14	0	14	0	1 △	21926	4	4
fbc-5458-5459	9857	4495	9791	4495	37	37	61	46	2 △	454	8	11
libtiff-ee2ce5-b5691a	171379	106867	14068	13454	15	15	15	15	1 ✓	13296	1	1
php-310991-310999	89230	18988	31084	16653	2	2	2	2	2 ✓	384	1	1
php-308734-308761	14692	4160	14692	4160	4	4	4	4	2 ✓	5771	1	1
php-308262-308315	90431	10845	9137	8516	0	0	0	0	0 X	7191	-	-
php-307562-307561	31597	6997	13425	6997	1	0	1	0	1 ✓	4918	1	1
php-309579-309580	60351	11416	25400	11416	2	2	2	2	1 ✓	46	1	1
php-310011-310050	77671	16558	8160	7316	32	32	49	49	0X	30647	-	-
php-309688-309716	71633	15744	10699	6516	31	30	32	31	0X	8398	-	-
php-309516-309535	27098	6314	27098	6314	1	0	1	0	1 ✓	4000	1	1
php-307846-307853	22131	4757	21654	4757	1	0	1	0	1 ✓	3867	1	1
php-311346-311348	9799	3879	9799	3879	50	38	72	60	2 ✓	312	1	1
php-307914-307915	47988	15066	43285	15066	1	0	1	0	1 ✓	5748	1	1
php-309111-309159	52908	12232	29459	12232	10	1	10	1	1 △	24347	10	10
php-309892-309910	40758	9999	17455	9999	17	17	22	22	3 ✓	179	1	1

Table 5.7: SPR Per-defect Patch Generation Results

validated patch although it eventually finds a correct patch. "X" indicates that the algorithm fails to find any correct patch in 12 hours. The eleventh column (Correct Template Rank – In Space) presents the rank of the template that generates the first correct patch in the search space. The twelfth column (Correct Template Rank – In Plausible) presents the rank of the template among plausible templates. The last column (Correct Patch Rank in Plausible) presents the rank of the correct patch among all generated plausible patches.

The results show that for 5 out of the 19 defects (php-307562-307561, php-307846-307853, php-309516-309535, php-310991-310999, and php-307914-307915), all validated patches are correct. The results indicate that for these five defects, the supplied test suite is strong enough to identify correct patches within the Prophet search space. Therefore any patch generation order is sufficient as long as it allows the system to find a correct patch within 12 hours. In fact, all variants of Prophet find correct patches for these 5 defects.

For 10 of the remaining 14 defects, Prophet finds a correct patch as the first to validate. SPR, in contrast, finds a correct patch as the first to validate for 6 of these 14 defects. SPR empirically prioritizes candidate patches that change existing branch conditions above all other candidate patches in the search space (see Section 4.2.4). This rule conveniently allows SPR to find a correct patch as the first to validate for php-309579-309580, php-309892-309910, and php-311346-311348. See Section 5.3.3 for case studies of the four benchmark defects where Prophet outperforms SPR.

5.4.4 Condition Synthesis

The results in the fifth and seventh columns in Table 5.6 highlight the effectiveness of the validation algorithm with the condition synthesis. Among those evaluated partial instantiated templates with abstract conditions, only a significantly small portion is plausible. Up to 99% of the templates are discarded before Prophet even attempts to generate a concrete condition to repair the program. We observe a similar phenomenon in SPR (see Table 5.7), this is because Prophet and SPR differs only on the patch validation order.

We note that, for all defects except php-310991-310999, the condition generation algorithm is able to find an exact match for the recorded abstract condition values. For php-310991-310999, the correct generated condition matches all except one of the recorded abstract condition values. We attribute the discrepancy to the ability of the program to generate a correct result for both branch directions [102].

5.5 Search Space Extensions and Evaluation

This section presents a systematic quantitative analysis on the Prophet and SPR search space.

5.5.1 Methodology

Search Space Comparison: To explain why GenProg and AE generates correct patches for significantly less benchmark defects, we first compare the Prophet and SPR search space with the search space of GenProg and AE. We manually analyze the correct patch of each benchmark defect inside the Prophet and SPR search space and identify the modification operation that generates the patch. We also analyze whether the patch is inside the GenProg and AE search space or not.

Search Space Extension Implementation: We identify possible extensions to the Prophet and SPR search space to include correct patches for more defects. We implement three different extensions to our search space. With these extensions, we obtain in total 16 different search space configurations for Prophet and SPR. The extended Prophet and SPR search space contains correct patches for 24 benchmark defects (up from 19 defects).

Patch Generation: We run Prophet and SPR with each of the search space configuration for the 24 benchmark defects inside the extended search space. For each defect, we analyze the generated plausible patches for the defect to determine whether the patch is correct or incorrect.

Defect	Modification Type
php-307562-307561	Replace†
php-307846-307853	Add Init†
php-307914-307915	Replace†
php-308262-308315	Add Guard†
php-308734-308761	Guarded Control†
php-309111-309159	Copy
php-309516-309535	Add Init†
php-309579-309580	Change Condition†
php-309688-309716	Change Condition†
php-309892-309910	Delete
php-310011-310050	Copy and Replace†
php-310991-310999	Change Condition†
php-311346-311348	Redirect Branch†
libtiff-ee2ce5-b5691a	Add Control†
libtiff-d13be-ccadf	Change Condition†
gmp-13420-13421	Replace†
gzip-a1d3d4-f17cbd	Copy and Replace†
fbc-5458-5459	Change Condition†
libtiff-5b021-3dfb3	Replace†

Table 5.8: Modification Type of Prophet Correct Patches

5.5.2 Search Space Analysis

The Prophet search space contains correct patches for 19 defects. Table 5.8 classifies the modification type of the correct patches for these 19 defects. Note that for some defects, there are multiple correct patches. For such defects, here we consider the first validated correct patches. The first column (Defect) presents the defect id.

The second column (Modification Type) presents the modification type of the correct patch for each defect. "Add Control" indicates that the patch inserts a control statement with no condition. "Guarded Control" indicates that the patch inserts a guarded control statement with a meaningful condition. "Replace" indicates that the patch modifies an existing statement using value replacement to replace an atom inside it. "Copy and Replace" indicates that the patch copies a statement from somewhere else in the application using value replacement to replace an atom in the statement. "Add Init" indicates that the patch inserts a memory initialization statement. "Delete" indicates that the patch simply removes statements (this is a special case of the Conditional Guard modification in which the guard condition is set to false). "Redirect Branch" indicates that the patch removes one branch of an if statement and redirects all executions to the other branch (by setting the condition of the if statement to true or false). "Change Condition" indicates that the patch changes a branch condition in a non-trivial way (unlike "Delete" and "Redirect Branch"). "Add Guard" indicates that the patch conditionally executes an existing statement by adding an if statement to enclose the existing statement. A "†" in the second column indicates that all Prophet patches for this defect is outside the search space of GenProg and AE (for 17 out of the 19 defects, the Prophet correct patch is outside the GenProg and AE search space).

For php-307846-307853, php-308734-308761, php-309516-309535, and libtiff-ee2ce5b7-b5691a5a, the correct patches insert control statements or initialization statements that do not appear elsewhere in the source file. For php-307562-307561, php-307914-307915, gmp-13420-13421, gzip-a1d3d4-f17cbd, php-310011-310050, and libtiff-5b021-3dfb3 the Prophet patches change expressions inside the copied or replaced statements. For php-309579-309580, php-310991-310999, php-311346-311348, php-308262-308315, php-309688-309716, libtiff-d13be-ccadf, and fbc-5458-5459 the Prophet generated patches change the branch condition in a way which is not equivalent to deleting the whole statement. These patches are therefore outside the search space of GenProg and AE, which only copy and remove statements.

Our results show that the Prophet and SPR search space is much more effective than the GenProg and AE search space. Because the correct patches for these defects are outside the search space of GenProg and AE, these two systems will *never* be able to generate these patches no matter how long one runs the two systems.

5.5.3 Defects Outside Prophet Search Space

The Prophet search space contains correct patches for 19 defects. It is possible to extend the Prophet search space to cover more defects. Extending Prophet to consider more suspicious locations from the defect localization results would bring correct patches for one additional defect into the search space (lighttpd-2661-2662).

Extending the Prophet condition space to include comparison operations ($<, \leq, \geq, >$) would bring correct patches for an additional two defects into the search space (lighttpd-1913-1914 and python-70056-70059). Extending the repair space to include patches that apply two transformation schemas (instead of only one as in the current Prophet implementation) would bring correct patches for another two defects into the space (php-308525-308529 and gzip-3fe0ca-39a362). Extending the Copy and Replace schema instantiation space to include more sophisticated replacement expressions would bring correct patches for five more defects into the search space (php-311164-311141, libtiff-806c4c9-366216b, gmp-14166-14167, python-69934-69935, and fbc-5556-5557). Combining all four of these extensions would bring an additional six more defects into the search space (php-307687-307688, php-308523-308525, php-309453-309456, php-310108-310109, lighttpd-1948-1949, and gzip-3eb609-884ef6). Correct patches for the remaining 34 defects require changes to or insertions of at least three statements.

All of these extensions come with potential costs. The most obvious cost is the difficulty of searching a larger repair space. A more subtle cost is that increasing the search space may increase the number of plausible but incorrect patches and make it harder to find the correct repair. We will investigate several search space extensions in the remaining of this section.

5.5.4 Search Space Extensions

A rich search space is critical for the success of patch generation systems, but extending a search space comes with potential costs. The most obvious cost is the difficulty of searching a larger repair space. A more subtle cost is that increasing the search space may increase the number of plausible but incorrect repairs and make it harder to find the correct repair. We next investigate straightforward extensions to the Prophet and SPR search space.

We implement three extensions to the SPR and Prophet search spaces: considering more candidate program statements to patch, synthesizing more sophisticated conditions, and evaluating more complicated value replacement transformations.

Defect	Localization Rank	RExt	CExt
lighttpd-2661-2662	1926	No	No
lighttpd-1913-1914	280	No	Yes
python-70056-70059	214	No	Yes
python-69934-69935	136	Yes	No
gmp-14166-14167	226	Yes	No

Table 5.9: Search Space Extensions

More Program Statements to Patch: The baseline SPR and Prophet configurations consider the first 200 program statements identified by the error localizer. We modify SPR and Prophet to consider the first 100, 200, 300, and 2000 statements.

Condition Synthesis Extension (CExt): We extend the baseline SPR and Prophet condition synthesis algorithm to include the "<" and ">" operators and to also consider comparisons between two check expressions (e.g., $E < K$, $E_1 == E_2$, and $E_1 > E_2$, where E, E_1, and E_2 are check expressions and K is a check constant). In the rest of this chapter, we use "CExt" to denote this search space extension.

Value Replacement Extension (RExt): We extend the baseline SPR and Prophet replacement transformations to also replace a variable or a constant in the target statement with an expression that is composed of either 1) a unary operator and an atomic value (i.e., a variable or a constant) which appears in the basic block containing the statement or 2) a binary operator and two such atomic values. The operators that SPR and Prophet consider are "+", "-", "*", "==", "!=", and "&". In the rest of this chapter, we use "RExt" to denote this search space extension.

With all three search space extensions, the generated SPR and Prophet search spaces contain correct patches for five more defects (i.e., 24 defects in total) than the baseline search space. Table 5.9 summarizes these five defects. The first column (Defect) contains entries of the form X-Y-Z, where X is the name of the application that contains the defect, Y is the defective revision in the application repository, and Z is the reference fixed revision in the repository. The second column (Localization Rank) presents the error localization rank of the modified program statement in the correct patch for the defect. The third column (RExt) presents whether the correct patches for the defect require the RExt extension (value replacement extension) The fourth column (CExt) presents whether the correct patches for the defect require the CExt extension (condition synthesis extension).

Remaining Defects: The extended Prophet search space contains correct patches for 24 defects. It is possible to bring correct patches for 10 more defects (php-311164-311141, php-308525-308529, libtiff-806c4c9-366216b, fbc-5556-5557, php-307687-307688, php-308523-308525, php-309453-309456, php-310108-310109, lighttpd-1948-1949, and gzip-3eb609-884ef6) with three additional extensions: 1) Applying two transformation schemas instead of only one in Prophet and SPR, 2) Further extending Copy and Replacement schema to include more operators, 3) Further extending the condition synthesis to include more operators. Correct patches for the remaining 35 defects require changes to or insertions of at least three statements.

Search Space Configurations: We obtained in total 16 different search space configurations derived from all possible combinations of 1) working with the first 100, 200, 300, or 2000 program statements identified by the error localizer, 2) whether to enable value replacement extension (RExt), and 3) whether to enable condition synthesis extension (CExt).

5.5.5 Results of Search Space Extentions

We next present the experimental result summary on the 16 different search space configurations. php is an outlier with a test suite that contains an order of magnitude more test cases than the other applications. We therefore separate the php results from the results from other benchmarks. We present the result summary for all of the 24 defects for which any of the search spaces contains a correct patch. See Appendix A for the detailed results of each defect in all different search space configurations.

Table 5.10 presents a summary of the results for all of the benchmarks except php. Table 5.11 presents a summary of the results for the php benchmark. Each row presents patch generation results for SPR or Prophet with one search space configuration. The first column (System) presents the evaluated system (SPR or Prophet). The second column (Loc. Limit) presents the number of considered candidate program statements

System	Loc. Limit	Space Extension	Correct		Plausible & Blocked	Timeout	Space Size	Correct Rank	Plausible in 12h	Correct in 12h
			In Space	First						
SPR	100	No	4	1	7(3)	3(0)	20068.5	4614.0	8(2747)	4(5)
SPR	100	CExt	4	1	7(3)	3(0)	20068.5	4614.0	8(11438)	3(4)
SPR	100	RExt	4	1	7(3)	3(0)	21999.8	6004.8	8(2742)	4(5)
SPR	100	RExt+CExt	4	1	7(3)	3(0)	21999.8	6004.8	8(11192)	3(4)
SPR	200	No	6	2	7(4)	2(0)	46377.6	17889.5	9(2558)	6(8)
SPR	200	CExt	6	2	7(4)	2(0)	46377.6	17889.5	9(10823)	4(6)
SPR	200	RExt	7	2	7(4)	2(1)	52864.3	24759.9	9(3753)	6(8)
SPR	200	RExt+CExt	7	2	7(4)	2(1)	52864.3	24759.9	9(10855)	4(6)
SPR	300	No	6	1	8(5)	2(0)	73559.6	22960.0	9(2818)	6(8)
SPR	300	CExt	8	1	8(6)	2(1)	73559.6	30761.8	9(10237)	4(6)
SPR	300	RExt	8	1	8(6)	2(1)	82187.2	32951.4	9(2069)	7(8)
SPR	300	RExt+CExt	10	1	8(7)	2(2)	82187.2	37427.4	9(10455)	5(6)
SPR	2000	No	7	2	7(5)	2(0)	523753.8	157038.4	9(751)	5(6)
SPR	2000	CExt	9	2	7(6)	2(1)	523753.8	156495.1	9(6123)	4(5)
SPR	2000	RExt	9	2	7(6)	2(1)	574325.1	200996.7	9(657)	5(6)
SPR	2000	RExt+CExt	11	2	7(7)	2(2)	574325.1	192034.0	9(5831)	4(5)
Prophet	100	No	4	4	4(0)	3(0)	20068.5	589.2	8(2481)	4(5)
Prophet	100	CExt	4	3	5(1)	3(0)	20068.5	589.2	8(11901)	3(4)
Prophet	100	RExt	4	4	4(0)	3(0)	21999.8	520.5	8(2183)	4(5)
Prophet	100	RExt+CExt	4	3	5(1)	3(0)	21999.8	520.5	8(11595)	3(4)
Prophet	200	No	6	5	4(1)	2(0)	46377.6	11382.8	9(2579)	5(7)
Prophet	200	CExt	6	4	5(2)	2(0)	46377.6	11382.8	9(10969)	4(6)
Prophet	200	RExt	7	5	4(1)	2(1)	52864.3	19581.0	9(1939)	5(6)
Prophet	200	RExt+CExt	7	4	5(2)	2(1)	52864.3	19581.0	9(10928)	4(5)
Prophet	300	No	6	4	5(2)	2(0)	73559.6	11997.2	9(2555)	5(7)
Prophet	300	CExt	8	3	6(4)	2(1)	73559.6	14466.8	9(10948)	4(6)
Prophet	300	RExt	8	4	5(3)	2(1)	82187.2	25769.1	9(1548)	5(6)
Prophet	300	RExt+CExt	10	3	6(5)	2(2)	82187.2	25455.1	9(10886)	4(5)
Prophet	2000	No	7	4	5(3)	2(0)	523753.8	188588.4	9(1229)	5(7)
Prophet	2000	CExt	9	3	6(5)	2(1)	523753.8	156555.8	9(8208)	4(6)
Prophet	2000	RExt	9	3	6(5)	2(1)	574325.1	170715.4	9(1216)	5(6)
Prophet	2000	RExt+CExt	11	2	7(7)	2(2)	574325.1	148288.4	9(7919)	4(5)

Table 5.10: Patch Generation Results with Search Space Extensions (excluding php)

System	Loc. Limit	Space Extension	Correct		Plausible & Blocked	Timeout	Space Size	Correct Rank	Plausible in 12h	Correct in 12h
			In Space	First						
SPR	100	No	12	8	3(3)	2(1)	13446.5	4157.8	11(237)	9(14)
SPR	100	CExt	12	8	4(4)	1(0)	13446.5	4157.8	12(415)	10(12)
SPR	100	RExt	12	8	4(4)	1(0)	14026.7	4360.8	12(288)	11(15)
SPR	100	RExt+CExt	12	8	4(4)	1(0)	14026.7	4360.8	12(421)	10(12)
SPR	200	No	13	9	3(3)	1(1)	26512.0	7369.8	12(197)	10(15)
SPR	200	CExt	13	9	3(3)	1(1)	26512.0	7369.8	12(330)	10(13)
SPR	200	RExt	13	9	3(3)	1(1)	28158.2	7984.5	12(200)	10(15)
SPR	200	RExt+CExt	13	9	3(3)	1(1)	28158.2	7984.5	12(323)	10(13)
SPR	300	No	13	8	4(4)	1(1)	41859.8	10915.2	12(176)	9(14)
SPR	300	CExt	13	8	4(4)	1(1)	41859.8	10915.2	12(305)	9(12)
SPR	300	RExt	13	8	4(4)	1(1)	44631.1	12440.9	12(179)	9(14)
SPR	300	RExt+CExt	13	8	4(4)	1(1)	44631.1	12440.9	12(313)	9(12)
SPR	2000	No	13	5	2(2)	6(6)	327905.6	81570.5	7(58)	5(6)
SPR	2000	CExt	13	5	2(2)	6(6)	327905.6	81570.5	7(126)	5(6)
SPR	2000	RExt	13	5	3(3)	5(5)	356104.8	83997.9	8(59)	5(6)
SPR	2000	RExt+CExt	13	5	2(2)	6(6)	356104.8	83997.9	7(127)	5(6)
Prophet	100	No	12	8	3(3)	2(1)	13446.5	2599.4	11(279)	11(15)
Prophet	100	CExt	12	6	6(6)	1(0)	13446.5	2599.4	12(466)	11(11)
Prophet	100	RExt	12	9	3(3)	1(0)	14026.7	3433.8	12(327)	11(15)
Prophet	100	RExt+CExt	12	6	6(6)	1(0)	14026.7	3433.8	12(458)	11(11)
Prophet	200	No	13	10	3(3)	0(0)	26512.0	3522.1	13(285)	13(18)
Prophet	200	CExt	13	7	6(6)	0(0)	26512.0	3522.1	13(447)	12(13)
Prophet	200	RExt	13	10	3(3)	0(0)	28158.2	4504.4	13(299)	12(17)
Prophet	200	RExt+CExt	13	7	6(6)	0(0)	28158.2	4504.4	13(434)	12(13)
Prophet	300	No	13	10	3(3)	0(0)	41859.8	4319.6	13(280)	13(18)
Prophet	300	CExt	13	7	6(6)	0(0)	41859.8	4319.6	13(425)	12(13)
Prophet	300	RExt	13	10	3(3)	0(0)	44631.1	5403.1	13(283)	12(17)
Prophet	300	RExt+CExt	13	7	6(6)	0(0)	44631.1	5403.1	13(422)	12(13)
Prophet	2000	No	13	7	2(2)	4(4)	327905.6	21118.6	9(117)	7(10)
Prophet	2000	CExt	13	4	4(4)	5(5)	327905.6	21118.6	8(153)	6(6)
Prophet	2000	RExt	13	6	2(2)	5(5)	356104.8	25168.5	8(104)	6(9)
Prophet	2000	RExt+CExt	13	4	4(4)	5(5)	356104.8	25168.5	8(183)	6(6)

Table 5.11: Patch Generation Results with Search Space Extensions (php only)

to patch under the configuration. The third column (Space Extension) presents the transformation extensions that are enabled in the configuration: No (no extensions, baseline search space), CExt (condition synthesis extension), RExt (value replacement extension), or RExt+CExt (both).

The fourth column (Correct In Space) presents the number of defects with correct patches that lie inside the full search space for the corresponding configuration. The fifth column (Correct First) presents the number of defects for which the system finds the correct patch as the *first* patch that validates against the test suite.

Each entry of the sixth column (Plausible & Blocked) is of the form X(Y). Here X is the number of defects for which the system discovers a plausible but incorrect patch as the first patch that validates. Y is the number of defects for which a plausible but incorrect patch blocked a subsequent correct patch (i.e., Y is the number of defects for which 1) the system discovers a plausible but incorrect patch as the first patch that validates and 2) the full search space contains a correct patch for that defect).

Each entry of the seventh column (Timeout) is also of the form X(Y). Here X is the number of defects for which the system does not discover any plausible patch within the 12 hour timeout. Y is the number of defects for which 1) the system does not discover a plausible patch and 2) the full search space contains a correct patch for that defect.

The eighth column (Space Size) presents the average number of candidate patch templates in the search space over all of the 24 considered defects. Note that SPR and Prophet may instantiate multiple concrete patches with the staged program repair technique from a patch template that contains an abstract expression (See Section 4.2). This column shows how the size of the search space grows as a function of the number of candidate statements to patch and the two extensions. Note that the CExt transformation extension does not increase the number of patch templates. Instead it increases the number of concrete patches which each patch template generates. The ninth column (Correct Rank) presents the average rank of the first patch template that generates a correct patch in the search space over all of those defects for which at least one correct patch is inside the search space. Note that the correct rank increases as the size of the search space increases.

Each entry of the tenth column (Plausible in 12h) is of the form X(Y). Here X is the number of defects for which the system discovers a plausible patch within the 12 hour timeout. Y is the sum, over the all of the 24 considered defects, of the number of plausible patches that the system discovers within the 12 hour timeout.

Each entry of the eleventh column (Correct in 12h) is of the form X(Y). Here X is the number of defects for which the system discovers a correct patch (blocked or not) within the 12 hour timeout. Y is the number of correct patches that the system discovers within the 12 hour timeout.

5.5.6 Plausible and Correct Patch Density

An examination of the tenth column (Plausible in 12h) in Tables 5.10 and 5.11 highlights the overall plausible patch densities in the search spaces. For the benchmarks without php, the explored search spaces typically contain hundreds up to a thousand plausible patches per defect. For php, in contrast, the explored search spaces typically contain tens of plausible patches per defect. We attribute this significant difference in the plausible patch density to the quality of the php test suite and its resulting ability to successfully filter out otherwise plausible but incorrect patches. Indeed, for three php defects, the php test suite is strong enough to filter out all of the patches in the explored search spaces except the correct patch.

An examination of the eleventh column (Correct in 12h) in Tables 5.10 and 5.11 highlights the overall correct patch densities in the explored search spaces. In sharp contrast to the plausible patch densities, the explored search spaces contain, on average, less than two correct patches per defect for all of the benchmarks including php. There are five defects with as many as two correct patches in any search space and one defect with as many as four correct patches in any search space. The remaining defects contain either zero or one correct patch across all of the search spaces.

5.5.7 Search Space Tradeoffs

An examination of the fourth column (Correct In Space) in Table 5.10 shows that the number of correct patches in the full search space increases as the size of the search space increases (across all benchmarks except php). But an examination of the fifth column (Correct First) indicates that that this increase does not translate into an increase in the ability of SPR or Prophet to actually find these correct patches as the first patch to validate. In fact, the ability of SPR and Prophet to isolate a correct patch as the first patch to validate reaches a maximum at 200 candidate statements with no extensions, then (in general) decreases from there as the size of the search space increases. For php, Table 5.11 shows that the number of correct patches in the space does not significantly increase with the size of the search space, but that the drop in the number of correct patches found as the first patch to validate is even more significant. Indeed, the 200+No Prophet configuration finds 10 correct patches as the first patch to validate, while the largest 2000+RExt+CExt configuration finds only four!

We attribute these facts to an inherent tradeoff in the search spaces. Expanding the search spaces to include more correct patches also includes more implausible and plausible but incorrect patches. The implausible patches consume validation time (extending the time required to find the correct patches), while the plausible but incorrect patches block the correct patches. This trend is visible in the Y entries in the sixth column in Table 5.10 (Plausible & Blocked) (these entries count the number of blocked correct patches), which generally increase as the size of the search space increases.

Tables 5.10 and 5.11 show how this tradeoff makes the baseline SPR and Prophet configurations perform best despite working with search spaces that contain fewer correct patches. Increasing the candidate statements beyond 200 never increases the number of correct patches that are first to validate. Applying the CExt and RExt extensions also never increases the number of correct patches that are first to validate.

Our results highlight two challenges that SPR and Prophet (and other generate and validate systems) face when generating correct patches:

- **Weak Test Suites:** The test suite provides incomplete coverage. The most obvious problem of the weak test suite is that it may accept incorrect patches. Our results show that (especially for larger search spaces) plausible but incorrect patches often block correct patches. For example, when we run Prophet with the baseline search space (200+No), there are only 4 defects whose correct patches are blocked; when we run Prophet with the largest search space (2000+RExt+CExt), there are 11 defects whose correct patches are blocked.

  A more subtle problem is that weak test suites may increase the validation cost of plausible but incorrect patches. For such a patch, SPR or Prophet has to run the patched application on all test cases in the test suite. If a stronger test suite is used, SPR and Prophet may invalidate the patch with one test case and skip the remaining test cases.

- **Search Space Explosion:** A large search space contains many candidate patch templates and our results show that it may be intractable to validate all of the candidates. For example, with the baseline search space (200+No), Prophet times out for only two defects (whose correct patches are outside the search space); with the largest evaluated search space (2000+RExt+CExt), Prophet times out for seven defects (whose correct patches are inside the search space).

Note that many previous systems [49, 55, 82, 104] neglect the weak test suite problem and do not evaluate whether the generated patches are correct or not. In contrast, our results show that the weak test suite problem is at least as important as the search space explosion problem. In fact, for all evaluated search space configurations, there are more defects for which SPR or Prophet generates plausible but incorrect patches than for which SPR or Prophet times out.

Search Space	SPR	Prophet	Random (SPR)	Random (Prophet)
100+No	52 / 3	38 / 4	65.0 / 1.4	65.0 / 1.4
100+CExt	65 / 2	53 / 3	73.0 / 1.0	73.0 / 1.0
100+RExt	52 / 3	38 / 4	65.1 / 1.4	65.1 / 1.4
100+RExt+CExt	65 / 2	53 / 3	73.0 / 1.0	73.0 / 1.0
200+No	59 / 4	45 / 5	73.5 / 2.7	76.5 / 2.1
200+CExt	66 / 3	54 / 4	78.1 / 1.7	78.1 / 1.7
200+RExt	59 / 4	45 / 5	76.2 / 2.1	75.9 / 2.1
200+RExt+CExt	66 / 3	54 / 4	77.8 / 1.7	77.8 / 1.7
300+No	60 / 5	50 / 5	80.2 / 2.0	78.2 / 2.1
300+CExt	75 / 2	63 / 3	83.9 / 1.3	83.2 / 1.4
300+RExt	62 / 4	50 / 5	81.4 / 1.4	79.9 / 2.1
300+RExt+CExt	75 / 2	63 / 3	85.2 / 1.1	84.4 / 1.2
2000+No	56 / 4	50 / 5	78.9 / 1.3	77.8 / 2.3
2000+CExt	72 / 2	63 / 3	86.6 / 0.7	83.2 / 1.4
2000+RExt	60 / 3	51 / 5	74.7 / 1.7	78.5 / 2.1
2000+RExt+CExt	72 / 2	64 / 3	82.6 / 1.1	84.4 / 1.3

Table 5.12: Costs and Payoffs of Reviewing the First 10 Generated Patches (excluding php)

Search Space	SPR	Prophet	Random (SPR)	Random (Prophet)
100+No	38 / 9	37 / 9	45.8 / 8.1	44.7 / 8.4
100+CExt	48 / 9	56 / 9	66.7 / 6.8	66.8 / 6.8
100+RExt	39 / 10	39 / 10	50.9 / 8.8	51.4 / 8.9
100+RExt+CExt	46 / 9	57 / 9	66.9 / 6.8	66.7 / 6.8
200+No	39 / 10	39 / 11	47.7 / 9.1	49.3 / 10.4
200+CExt	39 / 10	57 / 11	59.7 / 7.6	68.1 / 7.9
200+RExt	39 / 10	40 / 11	47.8 / 9.1	51.7 / 10.1
200+RExt+CExt	39 / 10	58 / 11	59.6 / 7.6	68.0 / 7.9
300+No	32 / 9	39 / 11	43.8 / 8.1	50.2 / 10.4
300+CExt	32 / 9	57 / 11	59.3 / 6.4	72.8 / 7.9
300+RExt	34 / 9	40 / 11	45.8 / 8.1	51.5 / 10.2
300+RExt+CExt	34 / 9	58 / 11	61.3 / 6.4	69.5 / 8.0
2000+No	7 / 5	25 / 7	16.7 / 4.4	37.4 / 6.3
2000+CExt	7 / 5	34 / 5	29.8 / 2.9	46.4 / 4.0
2000+RExt	9 / 5	17 / 6	18.7 / 4.4	29.3 / 5.3
2000+RExt+CExt	7 / 5	27 / 5	29.8 / 2.9	47.4 / 3.2

Table 5.13: Costs and Payoffs of Reviewing the First 10 Generated Patches (php only)

5.5.8 Prophet and SPR Effectiveness

We compare the effectiveness of the SPR and Prophet patch prioritization orders on different search space configurations by measuring the costs and payoffs for a human developer who reviews the generated patches to find a correct patch. We consider a scenario in which the developer reviews the first 10 generated patches one by one for each defect until he finds a correct patch. He gives up if none of the first 10 patches are correct. For each system and each search space configuration, we compute (over the 24 defects that have correct patches in the full SPR and Prophet search space) 1) the total number of patches the developer reviews (this number is the cost) and 2) the total number of defects for which the developer obtains a correct patch (this number is the payoff). We also compute the expected costs and payoffs if the developer examines the generated plausible SPR and Prophet patches in a random order. The raw data used to compute these numbers is available at Appendix A.

Tables 5.12 and 5.13 present these costs and payoffs. The first column presents the search space configuration. The second and third columns present the costs and payoffs for the SPR and Prophet patch prioritization orders; the fourth and fifth columns present the corresponding costs and payoffs for the random orders. Each entry is of the form X/Y, where X is the total number of patches that the developer reviews and Y is the total number of defects for which he obtains a correct patch. These numbers highlight the effectiveness of the SPR and Prophet patch prioritization in identifying correct patches within much larger sets of plausible but incorrect patches.

5.6 Discussion

Prophet automatically learns from past successful human patches to obtain a probabilistic, application-independent model of correct code. It uses this model to automatically generate correct patches for defects in real world applications. The experimental results show that, in comparison with previous patch generation systems, the learned information significantly improves the ability of Prophet to generate correct patches.

5.6.1 Prophet Hypothesis

The patch generation results are consistent with a key hypothesis of Prophet, i.e., that even across applications, correct code shares properties that can be learned and exploited to generate correct patches for incorrect applications. The results show that by learning properties of correct code from successful patches for different applications, Prophet significantly outperforms all previous patch generation systems on a systematically collected benchmark set of defects in large real-world applications.

We note that the applications in the training set share many characteristics with the benchmark applications on which we evaluate Prophet. Specifically, all of these applications are open source Linux applications, written in C, that can be invoked from the command line. It therefore remains an open question whether this hypothesis generalizes across broader classes of programs.

5.6.2 Important Features

The Prophet features capture interactions between the code in the patch and the surrounding code that the patch modifies. The learning algorithm of Prophet then identifies a subset of such interactions that characterize correct patches.

We inspected the feature weights of the learned model. Features with large positive weights capture positively correlated interactions. Examples of such interactions include 1) the patch checks a value that is used as a parameter in a nearby procedure call, 2) the patch checks a pointer that nearby code dereferences, and 3) the patch checks a value that was recently changed by nearby code. These features are positively correlated because the root cause of many defects is a failure to check for a condition involving values that the surrounding code manipulates. Successful patches often insert checks involving these values. Another example of a positively correlated interaction is an inserted function call that replaces one of the parameters with a pointer that nearby code manipulates. Such patches often correct defects in which the programmer forgot to perform an operation on the object that the pointer references.

Features with large negative weights capture negatively correlated interactions. Examples of such interactions include: 1) the patch changes the value of a variable whose value is also changed by nearby code in the same block, 2) the patch checks a global variable, and 3) the patch inserts a call to a function that nearby code in the same block also calls. These features are negatively correlated because they often correspond to redundant, irrelevant, or less organized program logic that rarely occurs in successful patches.

While none of these features is, by itself, able to definitively distinguish correct patches among candidate patches, together they deliver a model that makes Prophet significantly better at finding correct patches than the heuristic rules in SPR.

5.6.3 Search Space Tradeoff

The results in Section 5.5.7 show that straightforwardly enlarging the search space beyond the SPR and Prophet baseline increases the number of defects that have a correct patch in the search space. But it does not increase the ability of SPR and Prophet to find correct patches for more defects — in fact, these increases often cause SPR and Prophet to find correct patches for *fewer* defects!

We attribute this phenomenon to the following tradeoff. Straightforwardly enlarging the search space also increases the number of candidate patches and may increase the number of plausible patches. The increased number of candidate patches consumes patch evaluation time and reduces the density of the correct patches in the search space. The increased number of plausible but incorrect patches increases the chance that such patches will block the correct patch (i.e., that the system will encounter plausible but incorrect patches as the first patches to validate).

This highlights the importance of better search space design for the further improvement of patch generation systems. Although the Prophet search space is much more effective than the GenProg and AE on our benchmark applications, it still relies on a set of manually designed transformation schemas. The next chapter will present techniques that automatically infer code transforms and search spaces from human patches to more efficiently navigate the tradeoff space of the search space design. Our evaluation results in Chapter 7 show that the inferred code transforms navigate the search space tradeoff more efficiently and therefore generate more correct patches than manually crafted rules.

Chapter 6

Learning Universal Patching Strategies with Genesis

All previous generate and validate systems including Prophet work with a set of manually crafted transforms [53, 55, 58, 59, 82, 93, 103, 104]. This approach limits the system to fixing only those defects that fall within the scope of the transforms that the developers of the patch generation system can conceive. The results in Section 5.5 show that there is an inherent trade-off between the search space coverage and tractability. Straightforwardly enlarging those manually crafted transforms often does not improve the patch generation results — it may even cause the patch generation system to produce fewer correct patches.

This limitation is especially unfortunate given the widespread availability (in open-source software repositories) of patches developed by many different human developers. Together, these patches embody a rich variety of different patching strategies developed by a wide range of human developers, and not just the patch generation strategies encoded in a set of manually crafted transforms from the developers of the patch generation system.

This chapter presents a new patch generation system for Java programs, Geneiss, that processes sets of human patches to automatically infer code transforms and search spaces for automatic patch generation. To the best of my knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from successful patches.

Transforms: Each Genesis transform has two *template abstract syntax trees (ASTs).* One template AST matches code in the original program. The other template AST specifies the replacement code for the generated patch. Template ASTs contain *template variables*, which match subtrees or subforests in the original or patched code. Template variables enable the transforms to abstract away patch- or application-specific details to capture common patch patterns implemented by multiple patches drawn from different applications.

Generators: Many useful patches do not simply rearrange existing code and logic; they also introduce new code and logic. Genesis transforms therefore implement *partial pattern matching* in which the template AST for the patch contains free template variables that are not matched in the original code. Each of the free template variables is associated with a *generator*, which systematically generates new candidate code components for the free variable. This new technique, which enables Genesis to synthesize new code and logic in the candidate patches, is essential to enabling Genesis to generate correct patches.

Transform Generalization: The novel definition of transforms and generators in Genesis does not only provide an expressive way to capture common patching strategies used by human developers. It also enables a powerful generalization algorithm to automatically obtain transforms from a set of human patches. The generalization algorithm summarizes the common AST structures and the developer modifications of the supplied human patches. The algorithm produces a generalized transform 1) that covers all of the human patches it derives and 2) that is as restrictive as possible to limit the number of candidate patches the transform would generate.

Search Space Inference with ILP: A key challenge in patch search space design is navigating an inherent trade-off between coverage and tractability (see Section 5.5). On one hand, the search space needs to be large enough to contain correct patches for the target class of errors (coverage). On the other hand, the search space needs to be small enough so that the patch generation system can efficiently explore the space to find the correct patches (tractability).

Genesis navigates this tradeoff by formulating an integer linear program (ILP) whose solution maximizes the number of training patches covered by the inferred search space while acceptably bounding the number of candidate patches that the search space can generate (Section 6.2.5).

The ILP operates over a collection of subsets of patches drawn from a set of training patches. Each subset generalizes to a Genesis transform, with the final search space generated by the set of transforms that the solution to the ILP selects. Genesis uses a sampling algorithm to tractably derive the collection of subsets of patches for the ILP. This sampling algorithm incrementally builds up larger subsets of patches from smaller subsets, using a fitness function to identify promising candidate subsets (Section 6.2.4). Together, the sampling algorithm and final ILP formulation of the search space selection problem enable Genesis to scalably infer a set of transforms with both good coverage and good tractability.

Learning Patch Prioritization: Genesis implements a modified version of the learning algorithm in Prophet to obtain a probabilistic model to recognize correct patches. Genesis then uses the learned model to prioritize potentially correct patches in the inferred search space (Section 6.3.1). By combining both the new search space inference technique and the Prophet learning technique, Genesis leverages information in past successful human patches in two harmonic dimensions. The inference technique enables Genesis to operate on a productive search space automatically derived from patch generation strategies and patterns of human developers, while the learning technique enables Genesis to explore the inferred search space efficiently via prioritizing potentially correct patches in the space.

Sources. The previous version of this research presented in this chapter appeared in [62].

6.1 Motivating Example

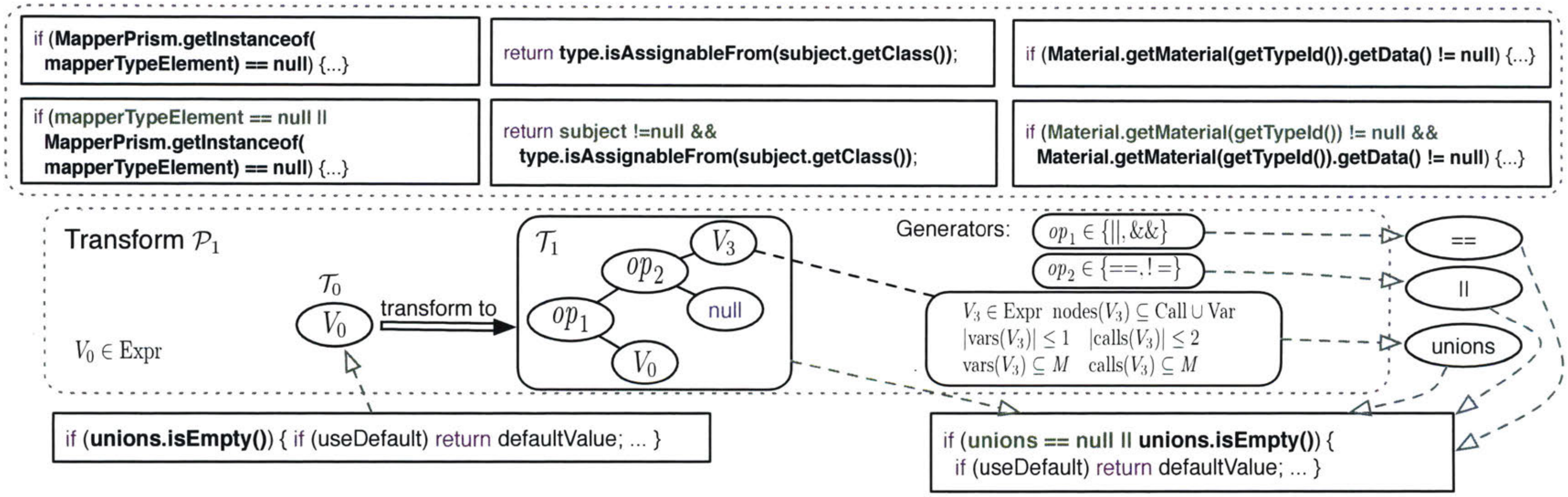
We next present, via an example, an overview of the Genesis transform inference algorithm. Genesis works with a training set of successful human patches to infer a set of patch generation transforms. In our example, the training set consists of 963 human patches collected from 356 github repositories.

Patch Sampling and Generalization: The Genesis inference algorithm works with sampled subsets of patches from the training set. For each subset, it applies a *generalization* algorithm to infer a *transform* that it can apply to generate candidate patches (Section 6.2.3). Figure 6-1 presents one of the sampled subsets of patches in our example: the first patch disjoins the clause `mapperTypeElement==null` to an if condition, the second patch conjoins the clause `subject!=null` to a return value, and the third patch conjoins the clause `Material.getMaterials(getTypeId())!=null` to an if condition. These patches are from three different applications, specifically mapstruct [19] 6d7a4d, modelmapper [21] d85131, and Bukkit [2] f13115. Genesis generalizes these patches to infer the transform $\mathcal{P}_1$ in Figure 6-1. When applied, $\mathcal{P}_1$ can generate all of the sampled three patches as well as other patches for other applications.

Template Anatomy: Each transform has a *template.* In our example, the template is $V_0 \Longrightarrow ((V_3)op_2(\text{null}))op_1(V_0)$ (Figure 6-1 presents this template in graphical form). The transform has an *initial template AST* $\mathcal{T}_0$, which matches a boolean expression V_0 in the unpatched program. V_0 must occur within a function body (if all of the training patches had modified if conditions, $\mathcal{T}_0$ would have reflected that more specific context).

The transform also has a *replacement template AST* $\mathcal{T}_1$, which replaces the matched boolean expression V_0 with a patch of the form $((V_3)op_2(\text{null}))op_1(V_0)$. Here V_3, op_2, and op_1 are *unmatched template variables.* Each such variable is associated with a *generator*, which enumerates candidate code components for the variable.

Generator Constraints: *Generator constraints* control the components that the generator will enumerate. The generator constraints for op_2 and op_1 ($op_2 \in \{==, !=\}$



```
if (MapperPrism.getInstanceof(
  mapperTypeElement) == null) {...}
```

```
return type.isAssignableFrom(subject.getClass());
```

```
if (Material.getMaterial(getTypeId()).getData() != null) {...}
```

```
if (mapperTypeElement == null ||
  MapperPrism.getInstanceof(
  mapperTypeElement) == null) {...}
```

```
return subject !=null &&
  type.isAssignableFrom(subject.getClass());
```

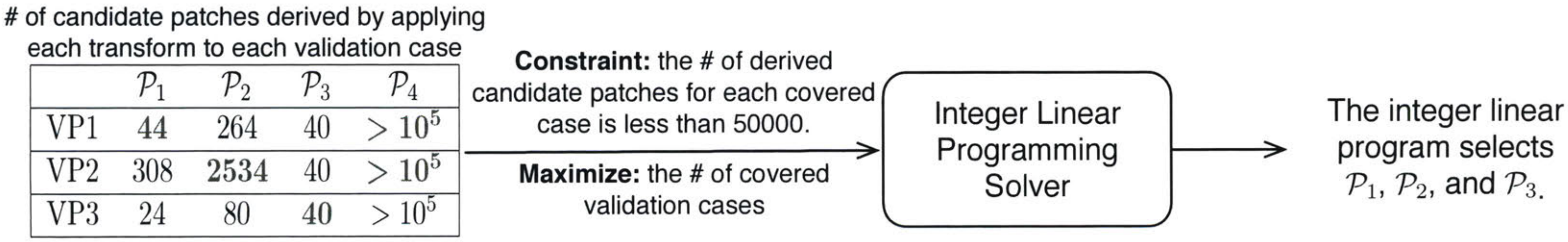
```
if (Material.getMaterial(getTypeId()) != null &&
  Material.getMaterial(getTypeId()).getData() != null) {...}
```

Transform $\mathcal{P}_1$
$\mathcal{T}_0$
V_0
transform to
$V_0 \in \text{Expr}$
$\mathcal{T}_1$
op_2
V_3
op_1
null
V_0
Generators:
$op_1 \in \{||, \&\&\}$
$op_2 \in \{==, !=\}$
$V_3 \in \text{Expr}$ $\text{nodes}(V_3) \subseteq \text{Call} \cup \text{Var}$
$|\text{vars}(V_3)| \leq 1$ $|\text{calls}(V_3)| \leq 2$
$\text{vars}(V_3) \subseteq M$ $\text{calls}(V_3) \subseteq M$
==
||
unions

```
if (unions.isEmpty()) { if (useDefault) return defaultValue; ... }
```

```
if (unions == null || unions.isEmpty()) {
  if (useDefault) return defaultValue; ... }
```

Figure 6-1: Example Inference and Application of a Genesis Transform. The training patches (original and patched code) are at the top, the inferred transform is in the middle, and the new patch that Genesis generates is at the bottom.



# of candidate patches derived by applying each transform to each validation case

	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$
VP1	44	264	40	$> 10^5$
VP2	308	2534	40	$> 10^5$
VP3	24	80	40	$> 10^5$

Constraint: the # of derived candidate patches for each covered case is less than 50000.
Maximize: the # of covered validation cases
Integer Linear Programming Solver
The integer linear program selects $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$.

Figure 6-2: Example of Using Integer Linear Programming to Select an Effective Set of Transforms.

and $op_1 \in \{\&\&, ||\}$) simply specify sets of operators to enumerate. The generator constraints for V_3 control the AST subtrees that the generator will enumerate for V_3. $V_3 \in \text{Expr}$ states that V_3 must be an expression. $\text{nodes}(V_3) \subseteq \text{Call} \cup \text{Var}$ states that V_3 can contain only method calls or variable references. $|V_3| \leq 2$ states that V_3 can contain at most 2 AST nodes.

$\text{vars}(V_3) \subseteq \text{M}$ states that any variables that appear in V_3 must also appear in the matched template AST V_0 (here M denotes the set of nodes in the original matched code). $|\text{vars}(V_3)| \leq 1$ states that at most 1 variable can appear in V_3. $\text{calls}(V_3) \subseteq \text{M}$ and $|\text{calls}(V_3)| \leq 2$ similarly constrain the method calls that may appear in V_3.

As these generator constraints illustrate, the Genesis patch generalization algorithm infers the *least general* Genesis transform that generates all of the sampled training patches. This strategy is critical for obtaining precisely targeted transforms that produce a tractable number of patches in the patch search space.

Candidate Transforms: Genesis repeatedly samples training patches to obtain the *candidate transforms* (from which Genesis will select the *selected transforms* that it uses for patch generation). In our example the candidate transforms include the previous transform $\mathcal{P}_1$ as well as a transform ($\mathcal{P}_2$) that adds a conditional (ternary) operator to guard the computation of an expression from NP defects, a transform ($\mathcal{P}_3$) that adds an if-guarded return or continue statement to skip the computation that triggers NP defects, and a transform $\mathcal{P}_4$ that replaces an arbitrary expression with a new expression. The new expression may contain binary operators, conditional operators, and up to six variables and six method calls from the enclosing function.

Not all of these transforms are equally useful. $\mathcal{P}_4$, for example, is an overly general transform that can generate an intractably large patch search space that Genesis cannot search effectively. $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$, on the other hand, are more targeted — because they were inferred from conceptually similar training patches, each generates a much smaller search space that nevertheless contains correct patches. And $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ effectively complement each other — their generated search spaces have relatively few patches in common.

Search Space Inference: To obtain an effective set of transforms, Genesis must discard overly general transforms such as $\mathcal{P}_4$ and include complementary and effectively targeted transforms such as $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$. Genesis drives the transform selection with a set of *validation patches* chosen from the training patches. Genesis starts by computing the number of validation patches that each candidate transform generates and the size of the search space that each candidate transform generates when applied to the pre-patch code for each validation patch.

The matrix in Figure 6-2 presents these numbers for the four candidate transforms $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$ and three validation patches VP1, VP2, and VP3 (in our example the validation patches are drawn from joda-time [17] revision bcb044, dynjs [15] revision 68df61, and orientdb [22] revision 51706f). Each number in the matrix is the number of candidate patches that a transform generates when applied to the pre-patch code of a validation patch. A **bold green number** indicates that a transform can generate the validation patch when applied to the pre-patch code of the patch. These numbers highlight the coverage vs. tractability tradeoff that the candidate patches present. With tractable search spaces, $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ all generate a single validation patch. $\mathcal{P}_4$, in contrast, generates two validation patches but at the cost of an intractably large search space.

Working with the information from the matrix, Genesis formulates an integer linear program (ILP) that maximizes the number of validation patches that the selected transforms can generate subject to the constraint that the total number of generated candidate patches from all selected transforms for each covered validation case is less than 5×10^4. In our example the ILP selects $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ as the selected transforms and excludes $\mathcal{P}_4$.

Patch Generation: For the NP defect from DataflowJavaSDK [4] revision c06125 (shown at the bottom of Figure 6-1), Genesis first uses a defect localization technique (Section 6.3.3) to produce a ranked list of potential statements to modify. The resulting ranked list includes the if condition shown at the bottom left of Figure 6-1. Genesis then applies all selected transforms, including $\mathcal{P}_1$, to the if condition to generate candidate patches.

Figure 6-1 shows how Genesis applies $\mathcal{P}_1$ to the if condition. Here the patch instantiates V_3 as the variable `unions`, op_2 as `==` and op_3 as `||` to disjoin the clause `unions == null` to the original if condition. The patch causes the enclosing function `innerGetOnly()` to return a predefined default value when `unions` is `null` (instead of incorrectly throwing a null pointer exception). This patch validates (produces correct outputs for all inputs in the DataflowJavaSDK JUnit [18] test suite), is correct, and matches the subsequent human developer patch for this defect.

6.2 Inference Algorithm

We next present the Genesis inference algorithm. Given a set of training pairs D, each of which corresponds to a program before a change and a program after a change, Genesis infers a set of transforms $\mathbb{P}$ that, working together, generate the search space of patches.

6.2.1 Preliminaries

Genesis works with abstract syntax trees (ASTs) of programs. We model the programming language that Genesis works with as a context free grammar (CFG) with abstract syntax trees (AST) as the parse trees for the CFG.

Definition 1 (CFG). *A* context free grammar (CFG) *G is a tuple $\langle N, \Sigma, R, s\rangle$ where N is the set of non-terminals, Σ is the set of terminals, R is a set of production rules of the form $a \rightarrow b_1 b_2 b_3 \ldots b_k$ where $a \in N$ and $b_i \in N \cup \Sigma$, and $s \in N$ is the starting non-terminal of the grammar. The language of G is the set of strings derivable from the start non-terminal: $\mathcal{L}(G) = \{w \in \Sigma^* \mid s \Rightarrow^* w\}$.*

Definition 2 (AST). *An* abstract syntax tree (AST) *T is a tuple $\langle G, X, r, \xi, \sigma\rangle$ where $G = \langle N, \Sigma, R, s\rangle$ is a CFG, X is a finite set of nodes in the tree, $r \in X$ is the root node of the tree, $\xi : X \rightarrow X^*$ maps each node to the list of its children nodes, and $\sigma : X \rightarrow (N \cup \Sigma)$ attaches a non-terminal or terminal label to each node in the tree.*

Definition 3 (AST Traversal and Valid AST). *Given an AST $T = \langle G, X, r, \xi, \sigma\rangle$ where $G = \langle N, \Sigma, R, s\rangle$, $\mathsf{str}(T)$ denotes the terminal string obtained via traversing T. T is a valid AST of G iff $\mathsf{str}(T) \in \mathcal{L}(G)$.*

We next define AST forests and AST slices, which we will use to present the Genesis inference algorithm. An AST forest is similar to an AST except it contains multiple trees and a list of root nodes. An AST slice is a special forest inside a large AST which corresponds to a list of adjacent siblings.

Definition 4 (AST Forest). *An AST forest T is a tuple $\langle G, X, L, \xi, \sigma\rangle$ where G is a CFG, X is the set of nodes in the forest, $L = \langle x_1, x_2, \ldots, x_k\rangle$ is the list of root nodes of trees in the forest, ξ maps each node to the list of its children nodes, and σ maps each node in X to a non-terminal or terminal label.*

Definition 5 (AST Slice). *An AST slice S is a pair $\langle T, L\rangle$. $T = \langle G, X, r, \xi, \sigma\rangle$ is an AST; $L = \langle r\rangle$ is a list that contains only the root node or $L = \langle x_{c_i}, \ldots, x_{c_j}\rangle$ is a list of AST sibling nodes in T such that $\exists x' \in X : \xi(x') = \langle x_{c_1}, \ldots, x_{c_i}, \ldots, x_{c_j}, \ldots, x_{c_k}\rangle$ (i.e., L is a sublist of $\xi(x')$).*

Given two ASTs T and T', where T is the AST before the change and T' is the AST after the change, Genesis computes AST difference between T and T' to produce an AST slice pair $\langle S, S'\rangle$ such that S and S' point to the sub-forests in T and T' that subsume the change. For brevity, in this section we assume $D = \{\langle S_1, S'_1\rangle, \langle S_2, S'_2\rangle, \ldots, \langle S_m, S'_m\rangle\}$ is a set of AST slice pairs, i.e., Genesis has already converted AST pairs of changes to AST slices.

Notation and Utility Functions: For a map M, $\mathrm{dom}(M)$ denotes the domain of M. $M[a \mapsto b]$ denotes the new map which maps a to b and maps other elements in $\mathrm{dom}(M)$ to the same values as M. $\emptyset$ denotes an empty set or an empty map.

$\mathsf{nodes}(\xi, L)$ denotes the set of nodes in a forest, where ξ maps each node to a list of its children and L is the list of the root nodes of the trees in the forest.

$\mathsf{inside}(S)$ denotes the set of non-terminals of the ancestor nodes of an AST slice S.

$\mathsf{nonterm}(L, X, \xi, \sigma, N)$ denotes the set of non-terminals inside a forest, where L is the root nodes in the forest, X is a finite set of nodes, ξ maps each node to a list of children nodes, σ attaches each node to a terminal or non-terminal label, and N is the set of non-terminals.

$\text{diff}(A, B)$ denotes the number of different terminals in leaf nodes between two ASTs, AST slices, or AST forests. If A and B differ in not just terminals in leaf nodes, $\text{diff}(A, B) = \infty$. $A \equiv B$ denotes that A and B are equivalent, i.e., $\text{diff}(A, B) = 0$.

6.2.2 Template AST Forest, Generator, Transforms

Template AST Forest: We next introduce the template AST forest, which can represent a set of concrete AST forests or slices. The key difference between template and concrete AST forests is that template AST forests contain template variables, each of which can match against any appropriate AST subtree or AST sub-forest.

Definition 6 (Template AST Forest). *A template AST forest $\mathcal{T}$ is a tuple $\langle G, V, \gamma, X, L, \xi, \sigma \rangle$, where $G = \langle N, \Sigma, R, s \rangle$ is a CFG, V is a finite set of template variables, $\gamma : V \rightarrow \{0, 1\} \times \text{Powerset}(N)$ is a map that assigns each template variable to a bit of zero or one and a set of non-terminals, X is a finite set of nodes in the subtree, $L = \langle x_1, x_2, \ldots, x_k \rangle$, $x_i \in X$ is the list of root nodes of the trees in the forest, $\xi : X \rightarrow X^*$ maps each node to the list of its children nodes, and $\sigma : X \rightarrow N \cup \Sigma \cup V$ attaches a non-terminal, a terminal, or a template variable to each node.*

For each template variable $v \in V$, $\gamma(v) = \langle b, W \rangle$ determines the kind of AST subtrees or sub-forests which the variable can match against. If $b = 0$, v can match against only AST subtrees not sub-forests. If $b = 1$, then v can match against both subtrees and sub-forests. Additionally, v can match against an AST subtree or sub-forest only if its roots have non-terminals in W.

Intuitively, each non-terminal in the CFG of a programming language typically corresponds to one kind of syntactic unit in programs at a certain granularity. Template AST forests with template variables enable Genesis to achieve a desirable abstraction over concrete AST trees during the inference. They also enable Genesis to abstract away program-specific syntactic details so that Genesis can infer useful transforms from changes across different applications.

Definition 7 (The $\models$ and $\models_{\texttt{slice}}$ Operators for Template AST Forests). *Figure 6-3 presents the formal definition of the operator* $\models$ *for a template AST forest* $\mathcal{T} = \langle G, V, \gamma, X, L, \xi, \sigma \rangle$. $\mathcal{T} \models \langle T, M \rangle$ *denotes that* $\mathcal{T}$ *matches the concrete AST forest* T *with the template variable bindings specified in* M, *where* M *is a map that assigns each template variable in* V *to an AST forest.*

Figure 6-3 also presents the formal definition of the operator $\models_{slice}$. *Similarly,* $\mathcal{T} \models_{slice} \langle S, M \rangle$ *denotes that* $\mathcal{T}$ *matches the concrete AST slice* S *with the variable bindings specified in* M.

The first rule in Figure 6-3 corresponds to the simple case of a single terminal node. The second and the third rules correspond to the cases of a single non-terminal node or a list of nodes, respectively. The two rules recursively match the children nodes and each individual node in the list.

The fourth and the fifth rules correspond to the case of a single template variable node in the template AST forest. The fourth rule matches the template variable against a forest, while the fifth rule matches the variable against a tree. These two rules check that the corresponding forest or tree of the variable in the binding map M is equivalent to the forest or tree that the rules are matching against.

Generators: Generators enumerate new code components:

Definition 8 (Generator). *A generator* $\mathcal{G}$ *is a tuple* $\langle G, b, \delta, W \rangle$, *where* $G = \langle N, \Sigma, R, s \rangle$ *is a CFG,* $b \in \{0, 1\}$ *indicates the behavior of the generator,* δ *is an integer bound for the number of tree nodes, and* $W \subseteq N$ *is the set of allowed non-terminals during generation.*

Generators exhibit two kinds of behaviors. If $b = 0$, the generator generates a sub-forest with less than δ nodes that contains only non-terminals inside the set W. If $b = 1$, the generator copies an existing sub-forest from the original AST tree with non-terminal labels in W and then replaces up to δ leaf nodes in the copied sub-forest.

Definition 9 (Generation Operator $\Longrightarrow$). *Figure 6-4 presents the formal definition of the operator* $\Longrightarrow$ *for a generator* $\mathcal{G}$. *Given* $\mathcal{G}$ *and an AST slice* $S = \langle T, L \rangle$ *as the context,* $\langle \mathcal{G}, S \rangle \Longrightarrow T'$ *denotes that the generator* $\mathcal{G}$ *generates the AST forest* T'.

$$\boxed{\begin{array}{ll} G = \langle N, \Sigma, R, s \rangle & \\ \mathcal{T} = \langle G, V, \gamma, X, L, \xi, \sigma \rangle & L = \langle x_1, x_2, \ldots, x_k \rangle \\ T = \langle G, X', L', \xi', \sigma' \rangle & L' = \langle x'_1, x'_2, \ldots, x'_{k'} \rangle \end{array}}$$

$$\frac{k = k' = 1 \qquad \sigma(x_1) = \sigma'(x'_1) \in \Sigma}{\mathcal{T} \models \langle T, M \rangle}$$

$$\frac{k = k' = 1 \qquad \sigma(x_1) = \sigma'(x'_1) \in N \qquad \langle G, V, \gamma, X, \xi(x_1), \xi, \sigma \rangle \models \langle \langle G, X', \xi'(x'_1), \xi', \sigma' \rangle, M \rangle}{\mathcal{T} \models \langle T, M \rangle}$$

$$\frac{k = k' > 1 \qquad \forall i \in \{1, 2, \ldots, k\} \left(\langle G, V, \gamma, X, \{x_i\}, \xi, \sigma \rangle \models \langle \langle G, X', \{x'_i\}, \xi', \sigma' \rangle, M \rangle \right)}{\mathcal{T} \models \langle T, M \rangle}$$

$$\frac{k = 1 \qquad \sigma(x_1) = v \in V \qquad M(v) \equiv T \qquad \gamma(v) = \langle 1, W \rangle \qquad (\cup_{i=1}^{k'} \sigma'(x'_i)) \subseteq (W \cup \Sigma)}{\mathcal{T} \models \langle T, M \rangle}$$

$$\frac{k = k' = 1 \qquad \sigma(x_1) = v \in V \qquad M(v) \equiv T \qquad \gamma(v) = \langle 0, W \rangle \qquad \sigma'(x'_1) \in (W \cup \Sigma)}{\mathcal{T} \models \langle T, M \rangle}$$

$$\frac{\mathcal{T} \models \langle \langle G, X', L', \xi', \sigma' \rangle, M \rangle}{\mathcal{T} \models_{slice} \langle \langle \langle G, X', r', \xi', \sigma' \rangle, L' \rangle, M \rangle}$$

Figure 6-3: Definitions of $\models$ and $\models_{\texttt{slice}}$

$$\boxed{\begin{array}{ll} G = \langle N, \Sigma, R, s \rangle & S = \langle T, L \rangle \\ T = \langle G, X, r, \xi, \sigma \rangle & T' = \langle G, X', L', \xi', \sigma' \rangle \end{array}}$$

$$\frac{|\mathsf{nodes}(\xi', L')| \leq \delta \qquad \mathsf{nonterm}(L', X', \xi', \sigma', N) \subseteq W}{\langle \langle G, 0, \delta, W \rangle, S \rangle \Longrightarrow T'}$$

$$\frac{\begin{array}{c} \exists x' \in X \; (L'' \text{ is a sublist of } \xi(x')) \\ \mathrm{diff}(\langle G, X, L'', \xi, \sigma \rangle, T') \leq \delta \qquad \forall x'' \in L' \; (\sigma'(x'') \in W) \end{array}}{\langle \langle G, 1, \delta, W \rangle, S \rangle \Longrightarrow T'}$$

Figure 6-4: Definition of the $\Longrightarrow$ for the Generator $\mathcal{G} = \langle G, b, \delta, W \rangle$

The first rule in Figure 6-4 handles the case where $b = 0$. The rule checks that the number of nodes in the result forest is within the bound δ and the set of non-terminals in the forest is a subset of W. The second rule handles the case where $b = 1$. The rule checks that the difference result forest and an existing forest in the original AST is within the bound and the root labels are in W.

Transforms: Finally, we introduce transforms, which generate the search space inferred by Genesis. Given an AST slice, a transform generates new AST trees.

Definition 10 (Transform). *A transform $\mathcal{P}$ is a tuple $\langle A, \mathcal{T}_0, \mathcal{T}_1, B\rangle$. $A \subseteq N$ is a set of non-terminals that denote the context where this transform can apply; $\mathcal{T}_0 = \langle G, V_0, \gamma_0, X_0, L_0, \xi_0, \sigma_0\rangle$ is the template AST forest before applying the transform; $\mathcal{T}_1 = \langle G, V_1, \gamma_1, X_1, L_1, \xi_1, \sigma_1\rangle$ is the forest after applying the transform; B maps each template variable v that appears only in $\mathcal{T}_1$ to a generator (i.e., $\forall v \in V_1 \setminus V_0$, $B(v)$ is a generator).*

Definition 11 ($\Longrightarrow$ and $\Longrightarrow_{\texttt{slice}}$ Operators). *Figure 6-5 presents the formal definition of the $\Longrightarrow$ and $\Longrightarrow_{slice}$ operators for a transform $\mathcal{P}$. $\langle \mathcal{P}, S\rangle \Longrightarrow T'$ denotes that applying $\mathcal{P}$ to the AST slice S generates the new AST T'. $\langle \mathcal{P}, S\rangle \Longrightarrow_{slice} S'$ denotes that applying $\mathcal{P}$ to the AST slice S generates the AST of the slice S'.*

Intuitively, in Figure 6-5 A and $\mathcal{T}_0$ determine the context where the transform $\mathcal{P}$ can apply. $\mathcal{P}$ can apply to an AST slice S only if the ancestors of S have all non-terminal labels in A and $\mathcal{T}_0$ can match against S with a variable binding map M. $\mathcal{T}_1$ and B then determine the transformed AST tree. $\mathcal{T}_1$ specifies the new arrangement of various components and B specifies the generators to generate AST sub-forests to replace free template variables in $\mathcal{T}_1$. Note that $\langle S, T'\rangle \rhd T$ denotes that the obtained AST tree of replacing the AST slice S with the AST forest T' is equivalent to T.

6.2.3 Transform Generalization

The generalization operation for transforms takes a set of AST slice pairs D as input and produces a set of transforms, each of which can at least generate the corresponding changes of the pairs in D.

$$\frac{\begin{array}{c} S = \langle\langle G, X, r, \xi, \sigma\rangle, L\rangle \\ A \subseteq \mathsf{inside}(S) \qquad \mathcal{T}_0 \models \langle S, M\rangle \qquad B = \{v_1 \mapsto \mathcal{G}_1, v_2 \mapsto \mathcal{G}_2, \ldots, v_m \mapsto \mathcal{G}_m\} \\ \forall_{i=1}^{m} \left(\langle \mathcal{G}_i, S\rangle \Longrightarrow T_i''\right) \qquad M' = \{v_1 \mapsto T_1'', v_2 \mapsto T_2'', \ldots v_k \mapsto T_m''\} \\ \mathcal{T}_1 \models \langle T', M \cup M'\rangle \qquad \langle S, T'\rangle \rhd T \qquad \mathsf{str}(T) \in \mathcal{L}(G) \end{array}}{\langle\langle A, \mathcal{T}_0, \mathcal{T}_1, B\rangle, S\rangle \Longrightarrow T}$$

$$\frac{\begin{array}{c} 1 \le i \le j \le k \\ S = \langle\langle G, X, r, \xi, \sigma\rangle, L\rangle \quad L = \langle x_i, \ldots, x_j\rangle \quad \xi(x') = \langle x_1, x_2, \ldots, x_k\rangle \\ T' = \langle G, X', L', \xi', \sigma'\rangle \quad L' = \langle x_1'', x_2'', \ldots, x_{k'}''\rangle \\ X \cap X' = \emptyset \qquad L'' = \langle x_1, \ldots, x_{i-1}, x_1'', x_2'', \ldots, x_{k'}'', x_{j+1}, \ldots, x_k\rangle \end{array}}{\langle S, T'\rangle \rhd \langle G, X \cup X', r, (\xi \cup \xi')[x' \mapsto L''], \sigma \cup \sigma'\rangle}$$

$$\frac{S' = \langle T', L'\rangle \qquad \langle \mathcal{P}, S\rangle \Longrightarrow T'}{\langle \mathcal{P}, S\rangle \Longrightarrow_{\texttt{slice}} S'}$$

Figure 6-5: Definition of $\Longrightarrow$ and $\Longrightarrow_{\texttt{slice}}$ for the Transform $\mathcal{P}$

Definition 12 (Generator Generalization). *Figure 6-6 presents the definition of the generalization function $\psi(D)$. Given a set of of AST slice pairs $D = \{\langle S_1, S_1'\rangle, \langle S_2, S_2'\rangle, \ldots, \langle S_m, S_m'\rangle\}$ from the same CFG grammar G, where S_i is the generation context AST slice and S_i' is the generated result AST slice, $\psi(D) = \{\mathcal{G}_1, \mathcal{G}_2, \ldots \mathcal{G}_k\}$ denotes the set of the generators generalized from D.*

In Figure 6-6, $\mathbb{A}$ is the formula for a generator that generates from scratch (i.e., $b = 0$) and $\mathbb{B}$ is the formula for a generator that generates via copying from the existing AST tree (i.e., $b = 1$). The formula $\mathbb{A}$ produces the generator by computing the bound of the number of nodes and the set of non-terminals in the supplied slices. The formula $\mathbb{B}$ produces the generator by computing 1) the bound of the minimum diff distance between each supplied slice and an arbitrary existing forest in the AST tree and 2) the set of non-terminals of the root node labels of the supplied slices.

Definition 13 (Transform Generalization). *Figure 6-7 presents the definition of $\Psi(D)$. Given a set of pairs of AST slices $D = \{\langle S_1, S_1'\rangle, \langle S_2, S_2'\rangle, \ldots, \langle S_m, S_m'\rangle\}$ where S_i is the AST slice before a change and S_i' is the AST slice after a change, $\Psi(D)$ is the set of transforms generalized from D.*

$$\boxed{\begin{array}{ll} G = (N, \Sigma, R, s) & D = \langle\langle S_1, S'_1\rangle, \langle S_2, S'_2\rangle, \ldots, \langle S_m, S'_m\rangle\rangle \\ \forall i \in \{1, 2, \ldots, m\} : & \\ S_i = \langle T_i, L_i\rangle & T_i = \langle G, X_i, r_i, \xi_i, \sigma_i\rangle \\ S'_i = \langle T'_i, L'_i\rangle & T'_i = \langle G, X'_i, r'_i, \xi'_i, \sigma'_i\rangle \\ L'_i = \langle x'_{i,1}, x'_{i,2}, \ldots, x'_{i,k'_i}\rangle & \end{array}}$$

$$\psi(D) = \begin{cases} \{\mathbb{A}, \mathbb{B}\} & \forall i \in \{1, \ldots, m\}, \forall j \in \{1, \ldots, k'_i\}, \sigma'(x'_{i,j}) \in N \\ \{\mathbb{A}\} & \text{otherwise} \end{cases}$$

where:

$\mathbb{A} = \langle G, 0, \max_{i=1}^{m} |\mathsf{nodes}(\xi'_i, L'_i)|, \bigcup_{i=1}^{m} \mathsf{nonterm}(S'_i)\rangle$

$\mathbb{B} = \langle G, 1, \max_{i=1}^{m} \mathbb{C}_i, \bigcup_{i=1}^{m} \bigcup_{j=1}^{k'_i} \{\sigma'(x'_{i,j})\}\rangle$

$\mathbb{C}_i = \min_{L''_i} \mathrm{diff}(\langle T_i, L''_i\rangle, S'_i), \exists x'' \in X_i,\ L''_i$ is a sublist of $\xi_i(x'')$

Figure 6-6: Definition of the Generator Inference Operator ψ

$$\begin{array}{l} \Psi(\langle\langle S_1, S'_1\rangle, \langle S_2, S'_2\rangle, \ldots, \langle S_m, S'_m\rangle\rangle) = \\ \{\langle \cap_{i=1}^{m} \mathsf{inside}(S_i), \mathcal{T}_0, \mathcal{T}_1, B\rangle \mid \\ \quad \langle \mathcal{T}_0, M\rangle = \Psi'(\langle S_1, S_2, \ldots, S_m\rangle, \emptyset), \\ \quad \langle \mathcal{T}_1, M'\rangle = \Psi'(\langle S'_1, S'_2, \ldots, S'_m\rangle, M), \\ \quad B = \{v_i \mapsto \mathcal{G}_i \mid \\ \qquad\qquad v_i \in \mathrm{dom}(M') \setminus \mathrm{dom}(\mathrm{M}), \\ \qquad\qquad M'(v_i) = \langle b_i, W_i, \langle S''_{i,1}, S''_{i,2}, \ldots, S''_{i,m}\rangle\rangle, \\ \qquad\qquad P_i = \{\langle S_1, S''_{i,1}\rangle, \langle S_2, S''_{i,2}\rangle, \ldots, \langle S_m, S''_{i,m}\rangle\}, \\ \qquad\qquad \mathcal{G}_i \in \psi(P_i)\}\} \end{array}$$

Figure 6-7: Definition of the Generalization Function Ψ

$\mathbb{S} = \langle S_1, S_2, \ldots, S_m \rangle$ $G = (N, \Sigma, R, s)$ x' is a fresh node v' is a fresh template variable

$\forall i \in \{1, 2, \ldots, m\}:$ $S_i = \langle T_i, L_i \rangle$ $L_i = \langle x_{i,1}, x_{i,2}, \ldots, x_{i,k_i} \rangle$ $T_i = \langle G, X_i, r_i, \xi_i, \sigma_i \rangle$ $c_i = \sigma_i(x_{i,1})$

$\Psi'(\mathbb{S}, M) =$	**Conditions for k and c**	**Other Conditions**
$\langle\langle G, \emptyset, \emptyset, \langle\rangle, \emptyset, \emptyset \rangle, M \rangle$	$\forall i \in \{1, \ldots, m\}\; k_i = 0$	
$\langle\langle G, \emptyset, \emptyset, \{x'\}, \langle x' \rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto d\}\rangle, M \rangle$	$d \in \Sigma \quad \forall i \in \{1, \ldots, m\}$ $k_i = 1 \quad c_i = d$	
$\langle\langle G, V, \gamma, X' \cup \{x'\}, \langle x' \rangle,$ $\xi'[x' \mapsto L'], \sigma'[x' \mapsto d]\rangle, M' \rangle$	$d \in N \quad \forall i \in \{1, \ldots, m\}$ $k_i = 1 \quad c_i = d$	$\mathbb{S}' = \langle\langle T_1, \xi_1(x_{1,1})\rangle, \langle T_2, \xi_2(x_{2,1})\rangle, \ldots, \langle T_m, \xi_m(x_{m,1})\rangle\rangle$ $\Psi'(\mathbb{S}', M) = \langle \mathcal{T}, M' \rangle \quad \mathcal{T} = \langle G, V, \gamma, X', L', \xi', \sigma' \rangle$
$\langle\langle G, \{v\}, \{v \mapsto \langle 0, W \rangle\},$ $\{x'\}, \langle x' \rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto v\}\rangle, M \rangle$	$\exists i, i' \in \{1, \ldots, m\}$ $c_i \neq c_{i'}$	$M(v) = \langle 0, W, \langle S'_1, S'_2, \ldots, S'_m \rangle\rangle$ $\forall i \in \{1, \ldots, m\}\,(S_i \equiv S'_i)$
$\langle\langle G, \{v'\}, \{v' \mapsto \langle 0, W \rangle\},$ $\{x'\}, \langle x' \rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto v'\}\rangle, M' \rangle$	$\forall i \in \{1, \ldots, m\} \quad k_i = 1$ $\exists i', i'' \in \{1, \ldots, m\}$ $(c_{i'} \neq c_{i''})$	$\forall v \in \mathrm{dom}(M)$ $M(v) = \langle 0, W', \langle S'_1, S'_2, \ldots, S'_m \rangle\rangle \quad \exists i \in \{1, 2, \ldots, m\}\,(S_i \not\equiv S'_i)$ $W = N \cap (\cup_{i=1}^{m}\{\sigma_i(x_{i,1})\}) \quad M' = M[v' \mapsto \langle 0, W, \mathbb{S}\rangle]$
$\langle\langle G, \cup_{j=1}^{k} V_j, \cup_{j=1}^{k} \gamma_j,$ $\cup_{j=1}^{k} X_j, \langle r_1, r_2, \ldots, r_k \rangle,$ $\cup_{j=1}^{k} \xi_j, \cup_{j=1}^{k} \sigma_j \rangle, M'_k \rangle$	$\forall i \in \{1, \ldots, m\}$ $k_i = k'$ $k' > 1$	$M'_0 = M$ $\forall j \in \{1, \ldots, k\}$ $\mathbb{S}'_j = \langle\langle T_1, \langle x_{1,j}\rangle\rangle, \langle T_2, \langle x_{2,j}\rangle\rangle, \ldots, \langle T_m, \langle x_{m,j}\rangle\rangle\rangle$ $\Psi'(\mathbb{S}'_j, M'_{j-1}) = \langle\langle G, V_j, \gamma_j, X_j, \langle r_j \rangle, \xi_j, \sigma_j \rangle, M'_j \rangle$
$\langle\langle G, \{v\}, \{v \mapsto \langle 1, W \rangle\},$ $\{x'\}, \langle x' \rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto v\}\rangle, M \rangle$	$\exists i', i'' \in \{1, \ldots, m\}$ $k_{i'} \neq k_{i''}$	$M(v) = \langle 1, W, \langle S'_1, S'_2, \ldots, S'_m \rangle\rangle$ $\forall i \in \{1, 2, \ldots, m\}\,(S_i \equiv S'_i)$
$\langle\langle G, \{v'\}, \{v' \mapsto \langle 1, W \rangle\},$ $\{x'\}, \langle x' \rangle,$ $\{x' \mapsto \emptyset\}, \{x' \mapsto v'\}\rangle, M' \rangle$	$\exists i', i'' \in \{1, \ldots, m\}$ $k_{i'} \neq k_{i''}$	$\forall v \in \mathrm{dom}(M)$ $M(v) = \langle 1, W', \langle S'_1, S'_2, \ldots, S'_m \rangle\rangle \quad \exists i \in \{1, 2, \ldots, m\}\,(S_i \not\equiv S'_i)$ $W = N \cap (\cup_{i=1}^{m} \cup_{j=1}^{k_i} \{\sigma_i(x_{i,j})\})$ $M' = M[v' \mapsto \langle 1, W, \mathbb{S}\rangle]$

Figure 6-8: Definition of Ψ'

The formula for Ψ in Figure 6-7 invokes Ψ' twice to compute the template AST forest before the change $\mathcal{T}_0$ and the template AST forest after the change $\mathcal{T}_1$. It then computes B by invoking ψ to obtain the generalized generators for AST sub-slices that match against each free template variable in $\mathcal{T}_1$. Figure 6-8 presents the definition of Ψ'. Intuitively, Ψ' is the generalization function for template AST forests. $\Psi'(\mathbb{S}, M) = \langle \mathcal{T}, M' \rangle$ takes a list of AST slices $\mathbb{S}$ and an initial variable binding map M and produces a generalized template AST forest $\mathcal{T}$ and an updated binding map M'.

The first two rows in Figure 6-8 correspond to the formulas for the cases of empty slices and slices with a single terminal, respectively. The two formulas simply create an empty template AST forest or a template AST forest with a single non-terminal node. The third row corresponds to the formula for the case of a single non-terminal. The formula recursively invokes Ψ' on the list of children nodes of each slice and creates a new node with the non-terminal label in the result template AST forest as the root node.

The fourth and fifth rows correspond to the formulas for the cases where each slice is a single tree and the root nodes of the slice trees do not match. The fourth formula handles the case where there is an existing template variable in M that can match the slice trees. The formula creates a template AST forest with the matching variable. The fifth formula handles the case where there is no existing template variable in M that can match the slice trees. The formula creates a template AST forest with a new variable and updates the variable binding map to include the variable.

The sixth row corresponds to the formula for the case where each slice is a forest with the same number of trees. The formula recursively invokes Ψ' on each individual tree and combines the obtained template AST forests.

The seventh row corresponds to the formula for the case in which each slice is a forest, the forests do not match, and there is an existing template variable in M to match these forests. The formula therefore creates a template AST forest with the matching variable.

The eighth row corresponds to the formula for the case where the forests do not match and there is no template variable in M to match these forests. The formula creates a template AST forest with a new template variable and updates the variable binding map accordingly.

6.2.4 Sampling Algorithm

Given a training database D, we could obtain an exponential number of transforms with the generalization function Ψ described in Section 6.2.3, i.e., we can invoke Ψ on any subset of D to obtain a different set of transforms. Not all of the generalized transforms are useful. The goal of the sampling algorithm is to use the generalization function to systematically obtain a set of productive candidate transforms for the inference algorithm to consider.

Figure 6-9 presents the pseudo-code of our sampling algorithm. As a standard approach in other learning and inference algorithms to avoid overfitting, Genesis splits the training database into a training set D and a validation set E. Genesis invokes the generalization functions only on pairs in the training set D to obtain candidate transforms. Genesis uses the validation set E to evaluate generalized transforms. $\mathbb{W}$ in Figure 6-9 is a work set that contains the candidate subset of D that the sampling algorithm is considering to use to obtain generalized transforms. The algorithm runs five iterations. At each iteration, the algorithm first computes a fitness score for each candidate subset, keeps the top α candidate subsets, (we empirically set α to 1000 in our experiments) and eliminates the rest from $\mathbb{W}$ (lines 3-7). The algorithm then attempts to update $\mathbb{W}$ by augmenting each subset in $\mathbb{W}$ with one additional pair in D (see lines 8-10). $\text{fitness}(\mathbb{W}, \mathbb{S}, D, E)$ denotes the fitness score of the subset $\mathbb{S}$ based on the coverage and tractability of transforms generalized from $\mathbb{S}$.

6.2.5 Search Space Inference Algorithm

ILP Formulation: Given a set of candidate transforms $\mathbb{P}'$, the goal is to select a subset $\mathbb{P}$ from $\mathbb{P}'$ that successfully navigates the patch search space coverage vs. tractability tradeoff.

```
Input  : a training set of pairs of AST slices $D$ and a set of pairs of AST slices $E$
Output : a set of transforms $\mathbb{P}'$
$\mathbb{W} \leftarrow \{\{\langle S, S'\rangle, \langle S'', S'''\rangle\} \mid \langle S, S'\rangle \in D, \langle S'', S'''\rangle \in D, \langle S, S'\rangle \neq \langle S'', S'''\rangle\}$
for $i = 1$ to $5$ do
    $f \leftarrow \{\mathbb{S} \mapsto \text{fitness}(\mathbb{W}, \mathbb{S}, D, E) \mid \mathbb{S} \in \mathbb{W}\}$
    $\mathbb{W}' \leftarrow \{\mathbb{S} \mid \mathbb{S} \in \mathbb{W}, f(\mathbb{S}) > 0\}$
    Sort elements in $\mathbb{W}'$ based on $f$
    Select top $\alpha$ elements in $\mathbb{W}'$ with largest $f$ value as a new set $\mathbb{W}''$
    $\mathbb{W} \leftarrow \mathbb{W}''$
    if $i \neq 5$ then
        for $\mathbb{S}$ in $\mathbb{W}''$ do
            for $\langle S, S'\rangle$ in $D$ do
                $\mathbb{W} \leftarrow \mathbb{W} \cup \{\mathbb{S} \cup \langle S, S'\rangle\}$
$\mathbb{P}' \leftarrow \cup_{\mathbb{S} \in \mathbb{W}} \Psi(\mathbb{S})$
return $\mathbb{P}'$
```

Figure 6-9: Sampling Algorithm sample(D, E)

$$\mathbb{P}' = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k\} \qquad E = \{\langle S_1, S_1'\rangle, \ldots, \langle S_n, S_n'\rangle\}$$

$$C_{i,j} = |\{\mathsf{str}(T) \mid \langle \mathcal{P}_j, S_i\rangle \Longrightarrow T\}|$$

$$G_{i,j} = \begin{cases} 1 & \langle \mathcal{P}_j, S_i\rangle \Longrightarrow_{\texttt{slice}} S_i' \\ 0 & \text{otherwise} \end{cases}$$

$$\text{Variables: } x_i,\ y_i \quad \text{Maximize:} \eta \cdot \textstyle\sum_{i=1}^{n_0} x_i + \sum_{i=n_0+1}^{n} x_i \quad \text{Satisfy:}$$

$$\forall i \in \{1, \ldots, n\} : \zeta - (\zeta - \beta) \cdot x_i - \textstyle\sum_{j=1}^{k} C_{i,j} y_j \geq 0$$

$$\forall i \in \{1, \ldots, n\} : \textstyle\sum_{j=1}^{k} G_{i,j} y_j - x_i \geq 0$$

$$\forall i \in \{1, \ldots, n\} : x_i \in \{0, 1\}$$

$$\forall i \in \{1, \ldots, k\} : y_i \in \{0, 1\}$$

$$\text{Result Transform Set: } \mathbb{P} = \{\mathcal{P}_i \mid y_i = 1\}$$

Figure 6-10: Integer Linear Programming Formulas for Selecting Transforms Given a Set of Candidate Transforms $\mathbb{P}'$ and a Validation Set of AST Slice Pairs E

Figure 6-10 presents our formulation of the transform selection problem as an integer linear program (ILP). E is the set of training and validation patch pairs. In Figure 6-10, the first n_0 pairs are training patches (i.e. $\langle S_1, S_1'\rangle \ldots \langle S_{n_0}, S_{n_0}'\rangle$), the remaining pairs are validation patches. $C_{i,j}$ corresponds to the size of the search space derived from the j-th transform when applied to the i-th AST slice pair in E. $G_{i,j}$ indicates whether the space derived from the j-th transform contains the corresponding change for the i-th AST slice pair.

The variable x_i indicates whether the result search space covers the i-th AST slice pair. The variable y_i indicates whether the ILP solution selects the i-th transform.

The ILP optimization goal is to maximize the weighted sum of x, where $\eta = 0.1$ is a parameter that controls the weight of covering training patches. The intuition is to prioritize the coverage of validation patches because the validation patches are hidden during the generalization step.

The first group of constraints is for tractability. The i-th constraint specifies that the derived final search space size (i.e. $\Sigma_{j=1}^{k} C_{i,j} y_j$), when applied to the i-th AST pair in E, should be less than β if the space covers the i-th AST pair (i.e. $x_i = 1$) or less than ζ if the space does not cover the i-th AST pair (i.e. $x_i = 0$). We empirically set $\beta = 5 \times 10^4$ and $\zeta = 10^8$. The second group of constraints is for coverage. The i-th constraint specifies that if the final search space covers the i-th AST slice pair in E (i.e. $x_i = 1$), then at least one of the selected transforms should cover the i-th pair.

Inference Algorithm: Starting from a training set of AST slice pairs D, Genesis first removes 25% of the AST slice pairs from D to form the validation set E. It then runs the sampling algorithm to produce a set of candidate transforms $\mathbb{P}'$. It finally solves the above ILP with Gurobi [46], an off-the-shelf solver, to obtain the set of transforms $\mathbb{P}$ that generates the Genesis patch search space.

6.3 Implementation

We have implemented the Genesis inference algorithm for Java programs. We use the spoon library [77] to parse Java programs to obtain Java ASTs. We next discuss several extensions of the above inference algorithm for handling Java programs.

6.3.1 Integration with Patch Characteristic Learning

Genesis implements a modified version of the Prophet learning algorithm from Section 4.3 to prioritize potentially correct patches during the search space exploration. The learning algorithm in Genesis takes an inferred search space and a set of training human patches as the input. The algorithm produces the model parameter θ for the parameterized probabilistic model in Section 4.3. In Genesis, the inference algorithm and the learning algorithm share the same set of training human patches. For each

Commutative Operators	Is an operand or a result value of +, *, ==, !=, or integer bit operations
Binary Operators	Is a left operand, a right operand, or a result value of -, /, <, >, <=, >=, . (field access), <<, >>, or . (function call)
Unary Operators	Is an operand or a result value of -, ++ (increment), -- (decrement)
Enclosing Statements	Occurs in an assign/loop/return/if statement Is a function call parameter Is the callee of a call statement Is the callee of a call statement with no argument
Value Traits	Is 0, 1, boolean `true`, boolean `false`, an integer constant, a null constant, or a constant string literal

Table 6.1: Atomic Characteristics of Program Values for Java in Genesis

training patch pair for which the inferred search space can generate the reference correct patch, the learned model attempts to prioritize the correct patch against all other candidate patches.

The main difference between the learning algorithm in Genesis and the learning algorithm in Prophet is the set of extracted atomic characteristics. Table 6.1 presents the extracted atomic characteristics in Genesis. The extracted characteristics in Genesis include Java specific features to track function calls that do not have arguments. This is because such functions tend to be getter functions that behave differently than other functions. Genesis uses the same algorithm (see Section 4.3.4) to combine pairs of atomic characteristics into final features. In the current version, Genesis tracks in total 5280 features. Genesis has more features because the inferred search spaces in our experiments can have up to 85 different transforms (see Section 7.2), while Prophet only has five modification kinds. Each Genesis transform will introduce additional modification features in the learning algorithm.

Another difference is that in Genesis we set the geometric prior $\beta = 0.2$ in the model (in Prophet $\beta = 0.05$, see Section 4.3.1). This is because the defect localization algorithm in Genesis relies on the stack trace information, which produces more accurate results than the localization algorithm in Prophet.

6.3.2 Integration with Condition Synthesis

```
Input  : a test case $t$, the original program $p$, and a program location $\ell$ that
         corresponds to an expression (condition).
Input  : a list of side-effect-free expressions $\{e_1, \ldots, e_n\}$ which correspond to a group
         of $n$ candidate patches. The $i$-th patch replaces the expression at $\ell$ with $e_i$.
Output : a list of indexes of patches that pass $t$.
1  $\mathcal{T} \longleftarrow \emptyset$
2  $P \longleftarrow \emptyset$
3  $F \longleftarrow \emptyset$
4  for $i \in \{1, \ldots, n\}$ do
5      $f \longleftarrow 0$
6      for $j \in F$ do
7          $T \longleftarrow \mathcal{T}(j)$
8          if equivalent$(T, e_i, e_j)$ then
9              $f \longleftarrow 1$
10             break
11     if $f = 0$ then
12         $\langle f, T \rangle \longleftarrow$ runtest$(t, p, \ell, e_i)$
13         if $f = 1$ then
14             $F \longleftarrow F \cup \{i\}$
15             $\mathcal{T} \longleftarrow \mathcal{T}[i \to T]$
16         else
17             $P \longleftarrow P \cup \{i\}$
18 return $P$
```

Figure 6-11: The Condition Synthesis Algorithm in Genesis

Genesis also implements the condition synthesis technique in Prophet (see Section 4.2.3). Figure 6-11 presents the condition synthesis algorithm in Genesis. Note that runtest(t, p, ℓ, e_i) at line 12 denotes a utility function that runs the test case t on the candidate patch that replaces the original expression at ℓ in p with the new expression e_i. It returns a pair that contains the test result and the test execution trace. The test result is either 1 (i.e., failing) or 0 (i.e., passing). The test execution trace records the values of all available variables at the program location ℓ. equivalent(T, e_i, e_j) is a utility function that check whether e_i and e_j are equivalent if they are evaluated with the values recorded in the given trace T.

To better integrate with the automatically inferred code transforms, the Genesis implementation does not explicitly explore different branch combinations. Instead, Genesis groups candidate patches that manipulate the same expression (condition) together. Then for each failed validation run, Genesis instruments the run to record all available program values in the patched expression (condition). When a new candidate patch that manipulates the same expression arrives, Genesis first evaluates the patched expression (condition) in the new patch with the recorded program values in the failed run to check whether this new patch will evaluate to the same value, as shown in lines 6-10 in Figure 6-11. If so, Genesis prunes away this new patch without further validation because this patch will produce same results as the failed run.

One advantage of this alternative implementation is that it generalizes to all side-effect-free expression changes. Condition synthesis for branch condition expressions are just a special case of side-effect-free expressions. However, our experimental results show that the condition synthesis does not provide as much reduction as it provides in Prophet (see Section 7.5). This is because Genesis typically infers a large number of transforms and the inferred transforms tend to generate more patches with side-effects. In fact, we observe that the reduction is not enough to offset the instrumentation overhead and the condition synthesis causes Genesis to generate fewer correct patches. Therefore we turn off the condition synthesis technique in Genesis by default.

6.3.3 Defect Localization

Genesis is designed to work with arbitrary defect localization algorithms. Our current implementation starts with stack traces generated from test cases that trigger the null pointer or out of bounds defect. It extracts the top ten stack trace entries and discards any entries that are not from source code files in the project (as opposed to external libraries or JUnit). Suppose the extracted stack trace contains the statements s_1, s_2, ..., s_{10} in order. For each statement s, Genesis first computes $a(s)$, $b(s)$, and

$c(s)$ as follows:

$$\begin{aligned} a(s) &= \arg\min_i \left(\text{linedis}(s, s_i)\right) \\ b(s) &= \text{linedis}(s, s_{a(s)}) \\ c(s) &= \begin{cases} 1 & \text{s is if/while/for/try statements} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Note that $\text{linedis}(s, s')$ denotes the line distance between two program statements s and s'. Genesis then computes the suspiciousness score $f(s)$ of the statement s as follows. Genesis sorts all statements according to their scores.

$$f(s) = \max(0, 0.5 - (a(s) + b(s))/60 + c(s) \times 0.5)$$

Intuitively, for each entry it finds the corresponding line of code in a project source code file and collects that line as well as the 50 lines before and after that line of code. The line of code given by the first stack trace entry has a suspiciousness score of 0.5, with the score linearly decreasing to zero as the sum of the distance to the closest line of code from the stack trace and the rank of that line within the stack trace increases. Genesis prioritizes lines containing if, try, while, or for statements by adding 0.5 to their suspiciousness scores. The final scores are in the range of 0 to 1.

6.3.4 Handling Java Programs

Identifiers and Constants: The CFG for Java has an infinite set of terminals, because there are an infinite amount of possible variables, fields, functions, and constants. For the kind of generators that enumerate all possible AST forests (i.e., $b = 0$), Genesis does not generate changes that import new packages and or changes that introduce new local variables (even if a change introduces new local variables, it is typically possible to find a semantically equivalent change that does not). Therefore Genesis only considers a finite set of possible variables, fields, and functions.

Genesis also extends enumeration-based generators so that each generator has an additional set to track the constant values that the generator can generate. For the generation operator of a generator, Genesis will only consider finite constant values that are 0, 1, `null`, `false`, or any value that is inside the tracked set of the generator.

Many string constants in Java programs are text messages (e.g., the message in throw statements). These constants may blow up the set of the allowed constants in the transforms during the generalization process. To avoid this problem, Genesis detects such string constants and convert them to a special constant string — the specific string values are typically not relevant to the overall correctness of the programs.

Identifier Scope: Genesis exploits the structure of Java programs to obtain more accurate generators. Each enumeration-based generator (i.e., $b = 0$) tracks separate bounds for the number of variables and functions it uses inside the original slice, from the enclosing function, from the enclosing file, and from all imported files. For example, a generator may specify that it will only use up to two variables from the enclosing function in the generated AST forests. Similarly, each copy-based generator (i.e., $b = 0$) has additional flags to determine whether it copies code from the original code, the enclosing function, or the entire enclosing source file.

Semantic Checking: Genesis performs type checking in its implementation of the generation operators for generators and transforms. Genesis will discard any AST tree or AST forest that cannot pass Java type checking. Genesis also performs semantic checking to detect common semantic problems like undefined variables and uninitialized variables.

Code Style Normalization: Genesis has a code style normalization component to rewrite programs in the training set while preserving semantic equivalence. The code style normalization enables Genesis to find more common structures among ASTs of training patches and improves the quality of the inferred transforms and search spaces.

Chapter 7

Evaluation of Genesis Patch Generation System

This chapter evaluates Genesis on a set of 49 benchmark defects systematically collected from open source repositories. All of our experimental results and the replication packages for Genesis are also available at [11]. The evaluation consists of three parts:

- **Patch Generation:** We compare the patch generation results of Genesis with the results of PAR [49], the previous state-of-the-art Java patch generation system, which uses a set of manually crafted templates.

- **Inferred Transforms:** We manually examine the Genesis inferred transforms to understand the advantage of the inferred transforms over manually crafted templates in previous systems.

- **Patch Prioritization and Condition Synthesis:** We evaluate the impact of the patch prioritization learning and the condition synthesis technique in Genesis. We investigate how these techniques interact with the inferred transforms.

- **Comparison of Patch Prioritization and Condition Synthesis in Genesis and Prophet:** We compare the effect of these techniques in Genesis and Prophet. We discuss how the manually crafted search spaces and the inferred search spaces interact with patch prioritization and condition synthesis techniques.

7.1 Benchmark and Training Defects

We developed a script that crawled the top 1000 github Java projects (ranked by number of stars), a list of 50968 github repositories from the MUSE corpus [9], and the github issue database. The script crawls the repositories and issues and it collects a project revision if 1) the project uses the Apache maven management system [20], 2) we can use maven 3.3 to automatically compile both the current revision and the parent revision of the current revision (in the github revision tree) in our experimental environment, 3) we can use the spoon library [77] to parse the source code of both of the two revisions into AST trees, 4) the issue discussion or the commit message of the current revision contains certain keywords to indicate that the revision corresponds to a patch for null pointer (NP), out of bound (OOB), or class cast (CC) defects, and 5) the revision changes only one source file (because revisions that change more than one source file often correspond to composite changes and not just patches for NP, OOB, or CC defects).

For NP defects, the scripts search for keywords “null deref”, “null pointer”, “null exception”, and “npe”. For OOB defects, the scripts search for keywords “out of bounds”, “bound check”, “bound fix”, and ”oob”. For CC defects, the scripts search for keywords “classcast exception”, “cast check”, and “cast fix”. We manually inspected the retrieved revisions to discard revisions that do not correspond to fixing NP, OOB, CC defects. Note that we discard many repositories and revisions because we are unable to automatically compile them with maven, i.e., they do not support maven or they have special dependencies that cannot be automatically resolved by the maven system.

We focus on NP, OOB, and CC defects because we want to compare the patch generation results of Genesis with PAR, a previous patch generation system. PAR [49] contains manual transform templates for NP, OOB, and CC defects. In fact, a review

	NP	OOB	CC	Combined
Sampled Transforms	607	504	503	579
Selected Transforms	43	20	17	85
Inference Time	12h	7h	8h	42h

Table 7.1: Genesis Search Space Inference Results

of PAR system finds out that most reported patches in the PAR paper are generated by its manual templates for NPE and OOB defects [69].

In total we collected 1012 human patches from 372 different applications. These patches include 503 null pointer error (NPE) patches, 212 out of bound error (OOB) patches, and 303 class cast error (CCE) patches. Note that there are six patches whose revision commit logs contain keywords for two kinds of defects and we count them as patches for both of the two kinds.

7.2 Offline Inference and Learning

Search Space Inference: We ran the inference algorithm on all of the training patches together to infer transforms for patching all three classes of defects combined. We also ran the inference algorithm on the training patches for each class of defects to infer transforms for that specific class of defects. Table 7.1 presents the results. The first row "Sampled Transforms" presents the number of candidate transforms produced by the sampling algorithm for each search space. The second row "Selected Transforms" presents the number of selected transforms in each inferred search space. We note that the number of selected transforms (tens for the targeted search spaces to over eighty for the combined search space) is substantially larger than the number of manually generated transforms that previous search spaces work with [49, 53, 55, 58, 59, 82, 93, 103, 104]. This fact highlights the ability of the automated Genesis inference system to work effectively with large, detailed, and appropriately targeted sets of candidate transforms.

Patch Prioritization Learning: We ran the learning algorithm on the training patches for each inferred search space to learn a probabilistic model to rank candidate patches in the space. Genesis tracks in total 5380 features and the offline learning finishes in less than one hour for all search spaces.

7.3 Patch Generation

This section evaluates the patch generation results of Genesis. It compares the results of Genesis with the results of PAR [49], the previous state-of-the-art patch generation system for Java.

7.3.1 Methodology

PAR Template Implementation: PAR is the previous state-of-the-art patch generation system for Java program [49]. It was first published in International Conference of Software Engineering (ICSE) 2013 and won a best paper award of ICSE 2013 [49, 69]. PAR deploys a set of transform templates to fix bugs in Java programs, with the templates manually derived by humans examining real-world patches. We implemented the PAR templates for NP, OOB, and CC defects under our own framework (most of the reported PAR patches are generated by the PAR NP and OOB templates [69]). To circumvent any ambiguities in the PAR template descriptions, we implemented the templates, with our best efforts, to enable the templates to generate correct patches for as many benchmark defects as possible.

Patch Generation: We ran Genesis on all of the benchmark defects with 1) the inferred transforms for patching each class of defects, 2) the inferred transforms for patching all three classes of defects combined, and 3) the PAR templates. For each defect, Genesis produces (a possibly empty) ranked list of validated patches. We performed all of the patch generation experiments on Amazon EC2 m4.xlarge instances with Intel Xeon E5-2676 processors, 4 vCPU, and 16 GB memory. We collected all patches that validated within 5 hours.

Analyze Generated Patches: We next manually analyzed each benchmark defect along with the corresponding developer patch from the repository to understand the root cause of the defect. We then analyzed the ranked list of generated patches to

Errors	Genesis (Per Defect Type) First 1/5/10 (All)	Genesis First 1/5/10 (All)	PAR First 1/5/10 (All)
20 NP	11/13/13 (13)	7/12/12 (13)	8/8/8 (8)
13 OOB	3/4/4 (6)	5/5/5 (6)	4/4/4 (4)
16 CC	2/2/2 (3)	4/4/4 (5)	0/0/0 (0)
Total	16/19/19 (22)	16/21/21 (24)	12/12/12 (12)

Table 7.2: Genesis Patch Generation Results

identify the first correct patch in each list (if any). Note that in our experiments all of the generated patches that we identify as correct are semantically equivalent to the corresponding developer patch (and differ in at most error or log messages).

We note that, in principle, determining patch correctness can be difficult. We emphasize that this is not the case for the patches in our experiments. The correct behavior for all of the defects is clear, as is patch correctness and incorrectness. All of the generated patches that we identify as correct are semantically equivalent to the corresponding developer patch (and differ in at most error or log messages).

7.3.2 Experimental Results

Table 7.2 summarizes the patch generation results. See Appendix B for the detailed per defect experimental results of each search space. The first column of Tabel 7.2 presents the number and type of each class of defect. The next columns present patch generation results for Genesis working with 1) the transforms for patching each class of defect, 2) the transforms for patching all three classes of defects combined, and 3) the PAR templates. Each entry is of the form X/Y/Z (W), where X is the number of defects for which the first patch to validate is correct, Y is the number of defects for which one of the first 5 patches to validate is correct, Z is the number of defects for which one of the first 10 patches to validate is correct, and W is the number of defects for which one of the validated patches is correct.

Our results show that Genesis significantly outperforms PAR. Genesis with the combined search space generates correct patches for 13 NP, 6 OOB, and 5 CC defects, while PAR templates generate correct patches for only 7 NP and 4 OOB defects.

Many of the Genesis generated correct patches are outside the search space of the PAR templates (see Section 7.4).

One interesting phenomenon is that moving to the combined transforms increases the number of cases with correct patches by two. One explanation is that learning from training patches of all three kinds of defects enables Genesis to better capture code transforms shared by different kinds of defects. In fact, the targeted CC space fails to generate correct patches for two cases because the space does not include a transform that inserts try-catch block to ignore exceptions. This transform can potentially fix NP, OOB, or CC defects.

This highlights the ability of Genesis to infer meaningful transforms to form a diverse search space for multiple classes of defects from a single combined corpus of human patches. With its patch prioritization learning algorithm, Genesis is also able to navigate the diverse search space efficiently by prioritizing potentially correct patches among all candidate patches in the space.

Consistent with previous results [60], there are many more validated patches than correct patches — for the combined search space, there are 8 NP defects, 4 OOB defects, and 1 CC defect with more than 20 validated patches. The maximum numbers of validated patches for a single NP, OOB, or CC defect are 62, 166, and 74 validated patches, respectively.

Defect Localization Oracle: To isolate the effect of defect localization, we also run Genesis and PAR with an oracle that identifies, for each defect, the correct line of code to patch. With the oracle, the combined Genesis space generates correct patches for one more NP defect (HikariCP-ce4ff92) and one more OOB defect (RoaringMap-29c6d59). PAR does not generate any additional correct patches with the oracle.

Correct Patches: In general, the Genesis correct patches either 1) apply a standard defect-specific patch pattern that Genesis successfully inferred, or 2) modify an existing condition or expression in the unpatched program. Four of the NP patches insert a null pointer check that returns if the checked variable is null. Another three insert a null pointer check that conditionally executes existing code only if the checked variable

```
 1     public ResponseEntity<?> deleteItemResource(
 2       RootResourceInformation resourceInformation, @BackendId Serializable id)
 3       throws ResourceNotFoundException, HttpRequestMethodNotSupportedException {
 4       resourceInformation.verifySupportedMethod(HttpMethod.DELETE, ResourceType.ITEM);
 5   +   if ((resourceInformation.getInvoker().invokeFindOne(id)) == null) {
 6   +     throw new org.springframework.data.rest.webmvc.ResourceNotFoundException();
 7   +   }
 8       org.springframework.data.rest.core.invoke.RepositoryInvoker invoker
 9         = resourceInformation.getInvoker();
10       Object domainObj = invoker.invokeFindOne(id);
11       publisher.publishEvent(new BeforeDeleteEvent(domainObj));
12       invoker.invokeDelete(id);
13       publisher.publishEvent(new AfterDeleteEvent(domainObj));
14       return new ResponseEntity<Object>(HttpStatus.NO_CONTENT);
15     }
```

Figure 7-1: Genesis Correct Patch for spring-data-rest-aa28aeb. Add Lines 5-7.

is not null. Two NP patches throw an exception if the checked variable is null; Another inserts a null pointer check that executes generated code when the check fires.

Three of the OOB patches insert a check for either a variable less than zero or an object field equal to zero, then return a generated value (either null or zero) if the check fires. One of the remaining OOB patches conditionally executes existing code if an object field is not zero. Two of the correct CC patches insert a try/catch statement that catches and ignores class cast exceptions. One of the remaining CC patches changes the declared type of a local variable.

The remaining NP, OOB, and CC patches change existing conditions or expressions in various ways. For example, one OOB patch corrects an off by one defect by changing < to <=; another OOB patch changes a condition to compare an expression against a variable instead of against zero; one CC patch wraps an expression to convert its value into another type.

7.3.3 Case Studies

We next investigate the results of four representative benchmark defects.

spring-data-rest-aa28aeb: Figure 7-1 presents the correct patch generated by Genesis for spring-data-rest-aa28aeb. This patch is semantically equivalent to the developer patch. The patch modifies the function deleteItemResource(). A null pointer error occurs when the user attempts to remove an item resource that does not exist.

```
1    public int unpack (ISOComponent m, byte[] b) throws ISOException {
2        ...
3   +    if ((b.length) == 0) {
4   +      return 0;
5   +    }
6        if ((logger) != null)
7          evt.addMessage(org.jpos.iso.ISOUtil.hexString(b));
8        ...
9    }
```

Figure 7-2: Genesis Correct Patch for jPOS-df400ac. Add Lines 3-5.

The patch fixes this defect by inserting a branch condition to check the returned value of the function invokeFindOne(). If the returned value is null, the supplied resource does not exist. The inserted code therefore throws an appropriate exception.

Note that PAR is unable to generate this patch because PAR does not have any template that inserts throw statements. This patch also highlights the difference between the inferred transforms in Genesis and the transformation schemas in Prophet. The checked expression in this patch is a sequence of two method calls, while the Prophet transformation schemas do not attempt to generate any function call in the inserted conditions.

jPOS-df400ac: Figure 7-2 presents the correct patch generated by Genesis for jPOS-df400ac. This patch is ranked as the first generated patch and it is semantically equivalent to the developer patch. The patch modifies the function unpack() in Figure 7-2. An OOB error occurs when the supplied byte array parameter "b" is empty. The patch fixes this defect by inserting a branch condition to check the length of the byte array. If the length is zero, it returns zero immediately.

Note that PAR is unable to generate this patch because PAR templates only consider conditions that check index out of bound for OOB errors. This patch highlights the coverage of the Genesis inferred transforms – the inferred transforms cover significantly more patching strategies than manually crafted templates.

jade4j-114e886: Figure 7-3 presents the correct patch generated by Genesis for jade4j-114e886. This patch is ranked as the first generated patch and it is semantically equivalent to the developer patch. The map "o" at line 4 is obtained via statically casting from another map. Therefore the keys in "o" may take integer values, even

```
 1     protected String visitAttributes(
 2       JadeModel model, JadeTemplate template) {
 3       ...
 4       for (Map.Entry<String, String> entry : o.entrySet()) {
 5         Attr attr = new Attr();
 6  -      attr.setName(entry.getKey());
 7  +      attr.setName(this.attributeValueToString(entry.getKey()));
 8         attr.setValue(entry.getValue());
 9         newAttributes.add(attr);
10       }
11       ...
12     }
```

Figure 7-3: Genesis Correct Patch for jade4j-114e886. Modify Lines 6-7.

```
1     public DefaultClassDefiner run() {
2  -    one = new DefaultClassDefiner(cd);
3  +    if (cd != null)
4  +            one = new DefaultClassDefiner(cd);
5  +    else
6  +            one = new DefaultClassDefiner(DefaultClassDefiner.class.getClassLoader());
7       return (DefaultClassDefiner) DefaultClassDefiner.defaultOne();
8     }
```

Figure 7-4: Developer Patch for nutz-80e85d0. Add Lines 2 and 4-5.

though the first type parameter of "o" is "String". When an integer key is retrieved in the loop, a CC error occurs at line 6. The patch wraps the expression with a function call to convert the expression value to string.

This patch highlights the expressiveness of generators in Genesis. The patch is generated by a transform that replaces an arbitrary expression with a new expression associated with a generator. The generator enumerates expressions that contain program elements from the original expression plus an additional method call that appears in the same file. The transform therefore can generate candidate patches that insert a function call to wrap the original expression.

nutz-80e85d0: Figure 7-4 presents the developer patch for nutz-80e85d0. The variable "cd" may equal to null at line 3, which causes a NP error. The developer patch wraps the function call at line 3 with an if-else statement. If "cd" equals null, it calls the static method getClassLoader() instead to replace "cd".

Although the Genesis search space contains transforms that insert if-else statements to wrap existing statements, this patch is still outside the Genesis space. This

```
 1 + int first = buffer.length();
 2   // pad the buffer with adequate zeros
 3   for(int digit = 0; digit<mSize; ++digit) {
 4     buffer.append('0');
 5   }
 6   // backfill the buffer with non-zero digits
 7   int index = buffer.length();
 8   for( ; value>0; value /= 10) {
 9 -   buffer.setCharAt(--index, (char)('0' + value % 10));
10 +   char c= (char)('0' + value % 10);
11 +   if(--index<first) {
12 +     buffer.insert(first, c);
13 +   }
14 +   else {
15 +     buffer.setCharAt(index, c);
16 +   }
17   }
```

Figure 7-5: Developer Patch for commons-lang-52b46e7. Modify Lines 1 and 9-16.

is because generators in the Genesis transforms cannot synthesize the expression DefaultClassDefiner.class.getClassLoader(). This case demonstrates the challenge of navigating the search space tradeoff between the coverage and the tractability. Note that it is possible to train Genesis on a larger training set of human patches to infer a richer and larger search space to cover this defect. But Genesis would also need more advanced search algorithms (with potentially better patch prioritization) to efficiently explore the larger search space.

commons-lang-52b46e7: Figure 7-5 presents the developer patch for commons-lang-52b46e7. The original code sets the character in a string buffer at a specific index at line 9. If "index" is greater than the length of "buffer" at line 9, an out of bound error occurs. Unlike usual OOB patches where the OOB accesses are discarded, the correct patch inserts additional characters into the buffer if the index is out of the range. The correct patch modifies multiple statements at two different locations to implement this behavior.

This correct patch is outside the inferred search space of Genesis and, despite the fact that Genesis deploys a focused search space derived by learning from successful human patches, is likely to remain beyond the reach of Genesis or similar generate and validate systems. In our experience, patch search spaces that include patches generated by transforming three or more statements can easily become intractable to search. Augmenting the search space to include such patches, in the absence of other techniques designed to improve the tractability of the search space, can leave the generate and validate system unable to find the correct patch even if the search space contains the correct patch.

7.4 Genesis Transforms

For the combined search space, Genesis selects 85 transforms from a set of 579 sampled transforms. This section first presents an overview of these 85 transforms. We then compare the inferred transforms with the manually crafted templates in other systems.

7.4.1 Transform Overview

Transforms That Target Boolean Expressions: Many defects involve incorrect boolean expressions [31, 58, 59, 65]. It is therefore not surprising that many of the inferred transforms target boolean expressions. Specifically, 16 of the 85 transforms (including the example transform discussed in Section 6.1) conjoin or disjoin a generated subexpression to a boolean condition in the original program.

Conditional Execution: 13 transforms conditionally execute existing matched code. 7 of the 13 add an if statement to guard the existing code, while the remaining 6 of the 13 add a conditional expression (i.e., ?:) to guard the existing code. There are also composite transforms such as a transform that 1) matches the declaration and the initialization of a variable, 2) change the initialization to first assign a default null or zero value, and 3) add an if statement to guard the original initialization code to avoid null pointer or out of bound defects:

$$\begin{aligned}&V_2\ V_1 = V_0; \Longrightarrow\\ &V_2\ V_1 = V_3;\quad \text{if}(V_6\ op_5\ V_8)\{V_1 = V_0\},\ V_3 \in \{\text{null}, 0\},\ op_5 \in \{<, ! =\}\end{aligned}$$

Inserted If Then Else: Two transforms wrap existing code in an if then else statement. The generated condition tests for a previously unhandled case, generated code handles the case on one branch, and the transform places existing code in the other branch. One transform, for example, inserts a null pointer check and generates code to return an empty array if the check succeeds. Another generates an equality check and sets a variable to a different value on the new branch:

$$V_0 = V_1; \Longrightarrow \text{if}(V_3{=}{=}V_4)\{V_0 = V_6; \}\text{else}\{V_0 = V_1; \}$$

Inserted If Then: 15 transforms insert conditionally executed generated code — the condition tests for a previously unhandled case. The generated code executes when the case occurs. To handle the case, typically the generated code either 1) includes throw, return, break, or continue statements or 2) is copied from the existing code elsewhere. 6 of the 15 transforms directly check for various null pointer cases, for example:

$$V_0 \Longrightarrow \text{if}(V_1 == \text{null})\{V_4\}; V_0$$

Replace Code: 24 transforms replace existing code with newly generated code. The transforms differ in 1) the form of the code they replace and generate and 2) the generator constraints. There are also several transforms that replace most but not all of the existing code. The following method call transform, for example, replaces the invoked method and parameters, but keeps the original receiver:

$$V_2.V_1(V_0) \Longrightarrow V_2.V_6(V_5)$$

Try/Catch/Continue: One transform wraps existing code in a try construct with an empty catch block. Like failure-oblivious computing [90], the patch discards the exception and continues execution:

$$V_0 \Longrightarrow \text{try}\{V_0\}\text{catch}(V_2)\{V_3\}, \text{where } V_3 \text{ is empty or a return statement.}$$

Change Declared Type: One transform changes the declared type (and potentially also the initializer) of a variable declaration. This transform generates patches that eliminate class cast exceptions, specifically by moving the declared type up in the class hierarchy (V_2 is the original declared type and V_4 is the new declared type):

$$V_2\, V_1 = V_0 \Longrightarrow V_4\, V_1 = V_3$$

Insert Statements: One transform inserts a generated assignment statement into the code. Another transform copies a statement from the existing code in the same function and inserts the copied statement.

Other Transforms: Genesis also infers a variety of more specialized transforms that, for example, combine null check insertions with bound check insertions. It also infers 9 transforms that, in our judgement, are specific to the defects in the training set and are unlikely to be useful for other defects. Even though these transforms are unlikely to correct any other defects, because they are so specific, they impose negligible search overhead.

Discussion: In comparison with previous manually developed transforms [49, 58, 59], the Genesis transforms are more numerous, more diverse, and target a wider range of defects more precisely and tractably. Some transforms target specific defect classes such as off by one defects in for loops. Other transforms apply general templates (for example, replacing one expression with another expression), with the generator constraints controlling the enumeration to deliver a tractable search space. While the inferred templates often correspond to intuitive patch generation patterns that can correct a wide range of defects, the generator constraints typically more closely reflect the specific characteristics of the patches in the training set. For example, many generator constraints focus the transform on introducing instanceof checks (to patch class cast exceptions), null pointer checks (to patch null pointer defects), or checks involving comparison operators such as $<$, $\leq$, $>$, or $\geq$ (to patch out of bounds defects). As the above discussion highlights, the combination of transform inference and generator constraints enables Genesis to infer a rich, precisely targeted, but still tractable patch search space.

7.4.2 Comparison with PAR

The PAR NP templates include "add an if statement to guard an existing statement" and "add an if guarded return statement" with null check boolean conditions. For the 13 NP defects for which Genesis generates correct patches, two NP defect (error-prone-370933 and spring-data-rest-aa28aeb) are outside the PAR search space because the correct patches add "if (...) throw ..." statements. Three more NP defects (DataflowJavaSDK-c06125 , javaslang-faf9ac , and Activiti-3d624a) are outside the PAR search space because the correct patches change condition expressions in a non-trivial way that is not equivalent to adding an if-guard.

The PAR OOB templates include "add an if statement to guard an existing statement" and "add an if guarded return statement" with range check conditions. The templates also consider "increases or decreases a variable by one" and "add an if guarded assignment statement to enforce index lower and upper bounds of a variable". For six OOB defects for which Genesis generates correct patches, two OOB cases (jgit-929862 and jPOS-df400a) are outside the PAR search space because the correct patches change conditions in a way different from the standard range checks.

The PAR CC templates include "add an if statement to guard an existing statement" and "add an if guarded return statement" with `instanceof` type checks. They also include "change the casting type of a cast operator". These templates do not generate correct patches for any of the benchmark CC defects. For two of these five defects (jade47-114e88 and HdrHistogram-030aac), the correct patches change existing expressions in a way that is not equivalent to adding a type check guard. The correct patches for two more CC defects (htmlelements-bf3f27 and hamcrest-bean-84586d) insert a try-catch statement to catch and ignore class cast exceptions (the developers introduced these try-catch statements in the patched revision and these statements are still present in the latest revision of these repositories). The correct patch for the remaining defect (jade4j-dd4739) modifies the declared type of a local variable to avoid class cast exceptions.

7.4.3 Comparison with Prophet

Note that the Prophet transformation schema in Section 4.2 is designed for C programs. If we straightforwardly convert the Prophet transformation schemas into Java transforms, it would contain correct patches for 11 NP and 5 OOB defects.

The correct patches for two NP defect (error-prone-370933 and spring-data-rest-aa28aeb) are outside the Prophet search space because they add "if (...) throw ..." statements. The correct patch for one OOB defect is outside the original Prophet search space (but inside the extended space in Section 5.5) because the patch manipulates a condition with two variables instead of one variable. The Prophet search space cannot generate correct patches for any CC errors, because the space does not contain schemas to change variable types, add `instanceof` type checks, or add try-catch blocks.

7.4.4 Defects Outside Genesis Search Space

Genesis does not generate correct patches for 25 out of the 49 benchmark defects. For HikariCP-ce4ff92 and RoaringBitmap-29c6d59, the correct patches are inside the search space of Genesis but the defect localization algorithm does not correctly identify the locations to modify. For the remaining 23 defects, the correct patches are outside the search space of Genesis.

For 19 out of the 23 defects, the correct patches modify less than three statements. For 11 out of the 19 defects (nutz-80e85d0, checkstyle-aaf606e, jongo-f46f658, spring-hateoas-29b4334, coveralls-maven-plugin-20490f6, maven-shared-77937e1, spoon-48d3126, pebble-942aa6e, fastjson-c886874, joinmo-a5ee885, antlr4-9e7b131), Genesis contains transforms with template AST pairs that match correct patches but the generators inside these transforms are not general enough to generate the correct patches. For 8 out of the 19 defects (webmagic-ff2f588, HdrHistogram-db18018, named-regexp-82bdfeb, pdfbox-93c0b69, tree-root-fef0f36, spring-cloud-connectors-56c6eca, buildergenerator-d9d73b3, mybatis-3-809c35d), Genesis does not contain transforms with template AST pairs to match correct patches. We expect learning from a larger training set of human patches and increasing the tractability parameter β (see Section 6.2.5) will generate a richer and larger search space to cover these 19 defects. However, due to the inherent tradeoff between the coverage and the tractability, enlarging the search space always comes with a cost on the tractability. We would need more advanced search algorithms (with potentially even better patch prioritization) to generate correct patches for these 19 defects in the larger search space.

For the remaining 4 defects (javapoet-70b38e5, truth-99b314e, commons-lang-52b46e7, raml-java-parser-49aab8f), the correct patches modify three or more statements. We believe these 4 defects are outside the capability of any generate-and-validate system.

7.5 Patch Prioritization and Condition Synthesis

This section evaluates the impact of the patch prioritization learning and condition synthesis techniques in Genesis.

7.5.1 Methodology

Patch Prioritization Learning Evaluation: We ran a modified version of Genesis with the combined search space which does not use the patch prioritization learning to rank candidate patches. This modified version uses the following heuristic rules instead. It explores its search space in order to each of the suspicious statements in the ranked list returned by the error localization algorithm. For each transform, this version computes a cost score which equals to the average number of candidate patches the transform need to generate to cover a validation case. For each suspicious statement, this version prioritizes candidate patches that are generated by those transforms with lower cost scores. If a transform produces two candidate patches that modify the same location, those two patches will be explored in an arbitrary order. We then compare the patch generation results of this modified version with the results of the original

	Genesis First 1/5/10 (All)	Genesis (No PP) First 1/5/10 (All)	Genesis (With CS) First 1/5/10 (All)
20 NP	7/12/12 (13)	11/12/13 (13)	7/12/12 (13)
13 OOB	5/5/5 (6)	5/5/6 (6)	5/5/5 (6)
16 CC	4/4/4 (5)	3/3/3 (4)	2/2/2 (3)
Total	16/21/21 (24)	19/20/22 (23)	14/19/19 (22)
Mean Gen. Time	31.9m	53.7m	46.7m

Table 7.3: Patch Generation Results for Three Different Versions of Genesis.

version of Genesis to evaluate the effectiveness of the patch prioritization learning algorithm in Genesis.

Condition Synthesis Evaluation: Genesis by default does not turn on its condition synthesis reduction (see Section 6.3.2). We ran a modified version of Genesis which turns on the condition synthesis with the combined search space. We then compare the patch generation results of the two versions to evaluate the effectiveness of the condition synthesis reduction in Genesis.

7.5.2 Experimental Results

Table 7.3 summarizes the patch generation results of three different versions of Genesis. See Appendix B for the detailed per defect experimental results of each version. The first column of Table 7.3 presents the number and type of each class of defect. The next columns present patch generation results for 1) the original Genesis working with the transforms for patching all three classes of defects combined, 2) the modified version of Genesis with the same space and without the patch prioritization learning, and 3) the modified version of Genesis with the same space and the condition synthesis reduction, Each entry is of the form X/Y/Z (W), where X is the number of defects for which the first patch to validate is correct, Y is the number of defects for which one of the first 5 patches to validate is correct, Z is the number of defects for which one of the first 10 patches to validate is correct, and W is the number of defects for which one of the validated patches is correct.

The last row of Table 7.3 presents the mean patch generation time of the first correct patch for the 24 benchmark defects which the original Genesis version produces correct patches. If a modified version of Genesis does not produce correct patch for one of the 24 defects, we count the timeout limit value (i.e., five hours) as the generation time for the defect.

Our results show that the patch prioritization learning improves the capability of Genesis to efficiently explore its search space. The original version of Genesis generates correct patches for one more cases than the modified version with the learning turned off. With the patch prioritization learning, Genesis takes on average 22 fewer minutes (i.e., 68% speedup) to find the first correct patch.

Intriguingly, our results show that the condition synthesis in Genesis does not provide as much reduction as we observed in Prophet. It only prunes away less than 18% candidate patches on average (see Appendix B). In fact, the amount of reduction is not enough to offset instrumentation overhead for implementing it. The modified version of Genesis with condition synthesis runs slower and generates correct patches for two fewer benchmark defects than the original version.

7.6 Discussion

We next discuss the difference between Genesis and PAR. We also discuss differences and similarities in how the patch prioritization and condition synthesis techniques interact with the inferred search spaces in Genesis and Prophet.

7.6.1 Comparison of Genesis and PAR

Genesis outperforms PAR because there are many design choices and parameters for a transform (template) and it is difficult for human to tune them manually. For example, consider an inferred transform for null pointer (NP) defects $A \rightarrow$ `if (B!=null) {A}`, which adds an if statement to guard an existing statement A. There are many design questions for this transform: what is allowed in the checked expression B? Should we just allow local variables and constants or do we allow complicated expressions? If we allow complicated expressions, how we are going to bound the expressions? If A is already an if statement, should we consider to change its condition instead of adding a new if statement? All these questions correspond to design choices and parameters in the transform. Suboptimal choices often produce unproductive search spaces that do not contain enough useful patches or that are too large to explore. It is difficult for manually defined templates to capture these complexities.

Our experimental results highlight how automating transform inference can produce more effective patch search spaces. Automating the inference makes it possible to work with larger sets of more detailed transforms that, working together, can more effectively navigate the inherent tradeoff between coverage and tractability. In comparison with manual transforms, the inferred transforms can be more specific (with focused generator parameters). But because there are so many more inferred transforms (over eighty for the combined training set) togethcr they cover many more patch patterns and can successfully patch more defects. By automating transform inference and formulating the transform selection problem as an integer linear program, Genesis can effectively work with hundreds to thousands of candidate transforms to select tens of final transforms.

7.6.2 Patch Prioritization in Genesis and Prophet

Genesis combines the Prophet patch prioritization learning algorithm together with the inferred search space to learn universal correctness properties to prioritize correct patches in the space. Genesis uses the same learning algorithm as Prophet with a different set of atomic characteristics for handling Java programs (see Section 6.3.1). Note that the original set of atomic characteristics in Prophet is designed for handling C programs (see Section 4.3.4).

Our experimental results show that the patch prioritization algorithm enables Genesis to rank the first correct patches over 60% higher on average. It therefore enables Genesis to generate correct patches 68% faster and enables Genesis to generate one more correct patch. Note that we had similar results in Prophet – the patch prioritization algorithm enables Prophet to rank the first correct patch 52% higher on average. The results show that the patch prioritization algorithm works well for the Genesis context, although Genesis targets a different programming language and different classes of defects than Prophet and operates with an inferred search space rather than a manually crafted search space.

Our results highlight how the learned universal correctness properties complement with the learned universal patching strategies. Genesis first applies the learned patching strategies to generate a sophisticated search space derived from close to one hundred inferred code transforms. Genesis then applies the learned universal correctness properties, encoded as a discriminative probabilistic model, to prioritize correct patches in the space to guide the patch generation search. Because the Genesis search space is automatically inferred and may contain significantly more transforms than Prophet (e.g., 85 transforms in Genesis and seven transformation schemas in Prophet), it is infeasible to manually craft optimal rules to determine the patch prioritization order. Therefore exploiting the universal properties to automate the patch prioritization process is even more important.

To apply the learning algorithm to Java, we only redesigned the set of the extracted atomic characteristics. The rest of the learning algorithm remains the same. The results therefore also highlight the generalizability of the patch prioritization learning algorithm across different programming languages. The success of the learning algorithm in Genesis demonstrates that the existence of universal correctness properties is programming-language-independent.

7.6.3 Condition Synthesis in Genesis and Prophet

The condition synthesis technique in Genesis only prunes away less than 18% candidate patches on average. In fact, the pruning is not even enough to offset the instrumentation overhead of implementing the condition synthesis technique. On the other hand, the condition synthesis in Prophet prunes away over 90% candidate patches on average. The reason behind this phenomena is the structural difference between the Genesis search space derived from the inferred code transforms and the Prophet search space derived from the seven transformation schemas.

Firstly, because each of the Genesis code transforms is more specific than the Prophet transformation schemas, the Genesis search space tends to contain fewer redundant candidate patches. For example, an inferred Genesis code transform for null pointer errors considers only conditions that check a variable against null value, while the transformation schemas in Prophet consider conditions that check a variable against any runtime constant value. As a result, there are fewer pruning opportunities for the condition synthesis in Genesis than in Prophet.

Secondly, Genesis has a relatively large number of small code transforms. Prophet has seven large transformation schemas, while the combined search space of Genesis contains 85 transforms. Each Genesis transform tends to generate much fewer candidate patches than each Prophet transformation schema does so that the total search space size of Genesis is still tractable. The current condition synthesis technique does not perform reductions for two candidate patches generated from different transforms. The technique therefore may miss pruning opportunities if two candidate patches modify the same expression but they are generated by different transforms.

Lastly, the Genesis inferred search space is much more diverse and tends to generate more patches with function calls than the Prophet search space. Java programs also tend to use function calls (e.g., getter functions) more often than C programs in branch conditions. The condition synthesis cannot prune away such patches. Five out of the seven Prophet transformation schemas do not introduce any new function call in the generated patch. In contrast, 52 out of the 85 Genesis inferred transforms for the combined search space contain generators that can introduce new function calls.

Chapter 8

Future Directions and Realistic Expectations

This chapter discusses potential future directions of learning successful human patches to build generate-and-validate systems like Prophet and Genesis. This chapter also discusses realistic expectations and potential limitations.

8.1 Future Directions

By learning universal properties and patching strategies of past successful human patches, Prophet and Genesis efficiently explore productive search spaces to generate correct patches for large real-world applications. We identify the following future research directions to further enhance Prophet and Genesis.

More Sophisticated Learning Algorithm: It is possible to design learning algorithms with more sophisticated models to extract more human knowledge from existing code. For example, how to apply neural network models to programs is still an interesting open question. One challenge is how to design new neural network structures that are capable to capture program semantic properties. Another challenge is how to effective map various program elements into numeric vectors for neural networks. Yet another challenge is that sophisticated models like neural networks will require significantly more training patches to drive the learning.

Exploiting Natural Language Resrouces: Natural language resources like commit logs and program comments are associated with existing code in software repositories. Although these resources are originally designed to help human developers to understand programs, we can exploit them to help machine understand programs as well. One future direction is therefore to combine natural language processing techniques together with the program learning techniques presented in this dissertation. A key challenge is to find a way to establish meaningful semantic mappings from natural language elements like words and phrases to program elements like variables and functions.

More Training Patches: In our experiments, we train Prophet with 777 training patches; we train Genesis with more than 900 training patches. But Prophet and Genesis could benefit from even more training patches. One challenge of collecting more training patches from open source repositories is to automatically compile the source code in the collected repositories. In our Genesis experiments, around 80% of crawled projects are discarded because we are unable to compile the projects due to missing dependencies. In our Prophet experiments, compiling C programs is even more difficult and we have to manually write a script to compile the revisions in each of the eight collected training projects. As a future work, an automated tool to resolve those compilation dependencies would be immensely helpful.

Learning Runtime Information: The learning algorithms in Prophet and Genesis consider only static information of the training human patches. It is possible to extend the algorithms consider runtime information, for example, determining the patch prioritization and the code transforms based on the execution traces of a program on its test cases. One challenge is that such extended algorithms would require an automated infrastructure to collect training inputs and run all training programs. As we discussed above, even automatically compiling the collected programs is already a non-trivial challenge. How to build an infrastructure to further automatically run the collected programs is still an open question.

Faster Sampling Algorithm: In our experiments, the Genesis search space inference takes up to 42 hours to finish. A significant part of the time is spent on the sampling algorithm to select promising subsets of the training patches. The complexity of the sampling algorithm is cubic and it may not be fast enough if we operate with a larger set of training patches. One future direction is therefore to design a faster sampling algorithm that still selects productive subsets of training patches for the generalization process.

Synthesis Reduction Across Transforms: In our experiments, the condition synthesis technique does not provide as much reduction in Genesis as we observed in Prophet. One reason is that Genesis has many smaller transforms, while Prophet has several large transforms. The current condition synthesis technique only operates with the candidate patches per transform. One future direction is therefore to improve the technique to enable reduction across different transforms.

Stronger Test Suite: Stronger test suites will enable generate-and-validate systems to generate less plausible but incorrect patches. One future direction is to automatically improve the test suite via test generation techniques. The key challenge here is how to classify a generated test case as positive or negative. Unless the test case triggers runtime errors, it is typically difficult to determine whether the program runs correctly on the case or not, because the generated test case may have triggered the defect in the program and produced an incorrect output.

8.2 Realistic Expectations and Limitations

Genesis generates correct patches for 24 out of the 49 defects. With a more accurate defect localization algorithm, Genesis is able to generate correct patches for two additional defects. As we discussed in Section 7.4.4, inferring transforms with richer template AST pairs would bring correct patches for up to 11 additional defects into the Genesis search space. Inferring transforms with more general generators would bring correct patches for up to 8 additional defects into the search space. Therefore if we successfully implemented the above enhancements in Section 8.1 for Genesis, Genesis would be able to generate correct patches for up to 45 defects out of the evaluated 49 defects. Similarly as we discussed in Section 5.5.3, a richer set of transforms would cover correct patches for 35 out of the 69 Prophet benchmark defects. If we successfully integrate these enhancement together with the Genesis inference system into Prophet, Prophet would be able to generate correct patches for up to 35 defects for the Prophet benchmark set.

Despite incorporating the learned information, Prophet and Genesis like all other generate-and-validate systems are still search-based systems. As we discuss above (Sections 5.5.3 and 7.4.4), in our experience, it is typically infeasible for generate-and-validate systems that operate at the level of statement transformations to generate a correct patch for a defect if the patch modifies more than three statements. For the 49 defects in the Genesis evaluation, four out of the 49 require to modify more than three statements. For the 69 defects in the Prophet evaluation, 34 out of the 69 require to modify more than three statements. The benchmark defects in the Genesis evaluation tend to require smaller modifications, because those defects are NP, OOB, and CC errors and the developer patches typically modify one or two statements to handle the corner cases that trigger those errors.

Another inherent limitation of Prophet and Genesis, like all other generate-and-validate systems, is the generation of plausible but incorrect patches. If we incorporate more learned information into Prophet and Genesis, Prophet and Genesis will prioritize correct patches more often and use more productive search space. However, because of the lack of formal specifications, Genesis will not be able to provide correctness guarantees for the generated patches. One possible way to circumvent this limitation is to have human developers to review a generated patch like the standard code review process for human patches.

Chapter 9

Related Work

We first present related work in automatic patch generation. We then discuss program learning and inference techniques for other software engineering tasks like specification mining, program synthesis, and code completion. We also present related work in relevant areas including defect localization and code refactoring.

9.1 Automatic Patch Generation

Automatic patch generation is an active research topic in the software engineering and programming language communities. We classify automatic patch generation techniques into three categories, 1) generate-and-validate techniques, which work with a test suite of inputs, generate a set of candidate patches, then test the patched programs against the test suite to find a patch that validates [49, 55, 65, 73, 80, 82, 104], 2) specification-based techniques that leverage formal specifications to produce patches that enable a defective program to satisfy the specification [79, 92], and 3) targeted techniques that repair specific classes of universal defects such as null dereferences [61], out of bounds accesses [90], and infinite loops [28, 50].

Automatic patch generation has been applied to generate feedback for online programming education as well [30, 97]. Because the problem setting of fixing submitted student programs is substantially different from fixing defects in software applications, we discuss these feedback generation techniques separately.

9.1.1 Generate-and-validate Techniques

Two earliest generate-and-validate patch generation systems are ClearView [80] and GenProg [103]. Since then, researchers have developed many other generate-and-validate patch generation systems [31, 37, 49, 55, 65, 73, 82, 104].

ClearView: ClearView is a generate-and-validate system that observes normal executions to learn invariants that characterize safe behavior [80]. It deploys monitors that detect crashes, illegal control transfers and out of bounds write defects. In response, it selects a nearby invariant that the input that triggered the defect violates, and generates patches to apply one of manually defined repair actions to enforce the violated invariant. Subsequent executions enable ClearView to determine if 1) the patch eliminates the defect while 2) preserving desired benign behavior. ClearView generates patches that can be applied directly to a running program without requiring a restart.

ClearView was evaluated by a hostile Red Team attempting to exploit security vulnerabilities in Firefox [80]. The Red Team developed attacks that targeted 10 Firefox vulnerabilities and evaluated the ability of ClearView to automatically generate patches that eliminated the vulnerability. For 9 of these 10 vulnerabilities, ClearView is able to generate patches that eliminate the vulnerability and enable Firefox to continue successful execution [80].

ClearView sits on a different point in the design space than Genesis and Prophet. It exploits the runtime information of individual programs to infer invariants, while Genesis and Prophet exploit the static information of human patches across different projects to improve their search space and search algorithm. ClearView is effective at fixing security vulnerabilities in browser and server applications, because 1) it is possible to collect a large number of execution traces for these applications and 2) enforcing violated runtime invariants with the manually defined repair actions is typically an effective way to fix security vulnerabilities. Genesis and Prophet, on the other hand, can be applied to broader classes of defects (not just security vulnerabilities) and broader kinds of scenarios, even when the runtime information is limited.

The ClearView technique and the learning technique in this dissertation are in fact orthogonal to each other. As we discussed in Section 7.6, one promising future direction is to combine the power of ClearView and Genesis, building a generate-and-validate system to learn from both static and runtime information of human patches across different applications. One challenge of pursuing this direction is how to design a generalization algorithm that extracts universal information from runtime invariants of different applications. Another challenge is how to collect a large training corpus of applications that the learning technique can automatically compile and run.

GenProg, RSRepair, and AE: GenProg [55, 103] uses a genetic programming algorithm to search a space of patches, with the goal of enabling the application to pass all considered test cases. RSRepair [82] changes the GenProg algorithm to use random modification instead. AE [104] uses a deterministic patch search algorithm and uses program equivalence relations to prune equivalent patches during testing.

Our results in Chapter 2 show that, contrary to the design principle of GenProg, RSRepair, and AE, the majority of the reported patches of these three systems are implausible due to errors in the patch validation. Further semantic analysis on the remaining plausible patches reveals that despite the surface complexity of these patches, an overwhelming majority of these patches are equivalent to functionality elimination. The Kali patch generation system, which only eliminates functionality, can do as well (see Section 2.8).

One potential explanation for this result is that the GenProg, RSRepair, and AE search spaces do not contain correct patches for these defects (see Section 5.5.2). Another potential explanation is that the search algorithms in GenProg, AE, and RSRepair cannot prioritize potentially correct patches in the space. AE and RSRepair search patches in their spaces in arbitrary and random orders, respectively. GenProg uses a genetic algorithm with a fitness function to guide the search, but the fitness function only relies on the test case information. The fitness of a patch in GenProg is defined as the number of passing test cases. For those common scenarios where there are only few correlated negative test cases, the genetic algorithm behaves similar to a random search [82].

Genesis and Prophet differ from GenProg, RSRepair, and AE in that Genesis and Prophet learn and exploit information to guide the patch generation process. This additional source of information enables Genesis and Prophet to work on better search algorithms that prioritize potentially correct patches and better search spaces that contain more useful patches. Our results show that most of the correct patches Prophet and Genesis generate in our experiments are outside the search space of GenProg, RSRepair, and AE.

PAR: PAR [49] deploys a set of patterns to fix bugs in Java programs, with the patterns manually derived by humans examining multiple real-world patches. In general, the design philosophy of PAR is to include the most widely used human patterns for common types of defects to avoid search space explosion. The disadvantage of this approach limits PAR to fix only those defects that fall within the scope of those patterns – a study found that most of the reported PAR patches are generated by the PAR NP and OOB templates [69].

The transformation schemas in SPR and Prophet tend to produce a richer search space than PAR especially for patches that manipulate conditions. Because of the condition synthesis technique in SPR and Prophet, SPR and Prophet can still explore their relatively larger search space efficiently. Our results in Section 7.4 show that SPR and Prophet transformation schemas, if straightforwardly converted to handle Java programs, can generate correct patches for more defects in the Genesis benchmark set than PAR templates.

Instead of relying on human examining real-world human patches to summarize patterns, Genesis automatically infers code transforms from real-world patches. This automation enables Genesis to use a larger number of transforms with each of the transforms being more focused than manually crafted templates. Together the inferred transforms deliver a more productive trade-off between the search space coverage and tractability, enabling Genesis to generate significantly more correct patches than PAR (see Section 7.3). The inference algorithm in Genesis also provides the flexibility – to apply Genesis on a new kind of defects, the user can train Genesis on a set of human patches that fix the new defect kind.

NOPOL: NOPOL [31, 37] applies the angelic debugging technique [29] to locate conditions that, if changed, may enable defective JAVA program to pass the supplied test suite. It then uses an SMT solver to synthesize patches for such conditions. Compared to Prophet and Genesis, NOPOL only target defects that can be fixed with patches that manipulate conditions. This focus enables NOPOL to limit the number of candidate patches it generates and avoid the search space explosion.

Prophet transformation schemas generate a relatively larger search space than NOPOL with patches that modify both branch conditions and other kinds of statements. Prophet uses the learned information from human patches to guide the search process and efficiently explore the search space. The code transforms in Genesis are automatically inferred from human patches. For NP, OOB, and CC defects, the inferred transforms focus on patches that modify branch conditions but they also include transforms such as changing the type of a declared variable or adding a try-catch block. The inference algorithm enables Genesis to efficiently navigate the trade-off of the search space design. It enables Genesis to operate with different transforms based on the kinds of defects Genesis is applied to.

SemFix and Angelix: SemFix [73] replaces a potentially faulty expression in a program with a symbolic value, performs symbolic executions on the supplied test cases to generate symbolic constraints, and uses SMT solvers to find concrete expressions that enable the program to pass the test cases. Because SemFix converts the whole program into SMT formulas, it is difficult to scale SemFix to large programs. In the SemFix paper, SemFix is only evaluated on programs with less than ten thousands lines of code [73].

Angelix [65] combines the technique in SemFix with the condition synthesis algorithm in SPR and Prophet. Angelix finds a sequence of values for the replaced faulty expression to pass the test suites and then uses the SMT solvers to find a patch with a concrete expression that generates the value sequence. Because Angelix searches the target value sequence like the condition synthesis algorithm in SPR and

Prophet does, Angelix does not need to convert the whole program into SMT formulas and therefore avoids the scalability problem in SemFix.

Angelix is also evaluated on the GenProg benchmark set and it generates correct patches for ten defects (eight fewer than Prophet). Prophet outperforms Angelix because 1) Angelix only generates patches that modify expressions and 2) Prophet exploits additional information learned from human patches to guide the patch generation process. Note that the synthesis algorithm in Angelix can handle multiple faulty expressions. This capability enables Angelix to fix defects that Prophet cannot fix. Theoretically it would be possible to extend Prophet and Genesis to handle multiple expressions as well. In practice, this would complicate the design of the learning and inference algorithms in Prophet and Genesis.

Anti-patterns: Tan et al. proposed a technique to define a set of manually crafted anti-patterns to reduce the number of plausible but incorrect patches generated by generate-and-validate systems [101]. Each anti-pattern encodes ineffective patch strategies or dangerous modifications that typically generate incorrect patches. In general, the anti-pattern technique provides an alternative way to rank candidate patches. Instead of prioritizing potentially correct patches, the technique aims to remove ineffective patches in the search space.

Note that the learning algorithm in Prophet and Genesis also deprioritize ineffective patches. The algorithm can defect whether an universal feature often correlates with incorrect patches in the training set. If so, it will assign a negative weight value in the learned feature parameter vector, deprioritizing any candidate patch that contains this universal feature – this universal feature effectively becomes an anti-pattern. The advantage of Prophet and Genesis is that such anti-patterns are automatically learned, rather than manually encoded.

9.1.2 Leveraging Human Patches

Besides Prophet and Genesis, researchers have developed other patch generation techniques to leverage information in past successful human patches.

Code Transfer: CodePhage automatically locates a correct condition check in one application, then transfers that condition check to eliminate defects in another application [94]. CodePhage has been applied to eliminate otherwise fatal integer overflow, buffer overflow, and divide by zero errors. CodeCarbonCopy [96] further extends the CodePhage technique to transfer functionality implementations across applications. uScalpel [24] uses test-driven genetic programming to transfer code from a donor application to a recipient application.

The code transfer techniques rely on the existence of donor applications that already contain the exact program logic required to eliminate the defect. The techniques in this dissertation, in contrast, learn universal patching strategies and properties of successful patches to guide the exploration of an automatically inferred search space of newly synthesized candidate patches. CodePhage uses symbolic execution traces to accurately translate variable names and data structures between the namespaces of two applications. It therefore may produce better patches for a defect than Prophet and Genesis, if the defect can be fixed with a code transfer from available donor applications. But Prophet and Genesis can be applied to any defect, not just those defects that can be fixed with code transfers.

Patch Generation via Q&A Sites: Gao et. al. [44] propose to repair recurring defects by analyzing Q&A sites such as Stack Overflow. The proposed technique locates the relevant Q&A page for a recurring defect via a search engine query, extracts code snippets from the page, and renames variables in the extracted code snippets to generate patches. Prophet and Genesis are different from their technique, because Prophet and Genesis do not rely on the existence of exact defect repair logic on Q&A pages. Their technique may produce better patches for a defect than Prophet and Genesis if there are high quality Stack Overflow pages for the defect. But Prophet and Genesis can be applied to any defect, not just those defects that have existing or similar fixes in Stack Overflow.

History-driven Repair and ACS: Like Prophet, history-driven program repair [53] uses information from previous human patches to rank candidate patches generated by human-specified transforms. It classifies candidate patches into categories based on the shapes of the abstract syntax trees of the patches and then prioritize those categories that appear more often in the previous human patches. ACS [106] ranks candidate variables in condition patches based on the control dependency and the data dependency of the variables in the patch context.

Prophet and Genesis differ from these two systems in that Prophet and Genesis use a machine learning algorithm instead of deterministic rules to extract universal properties of successful patches. This enables Prophet and Genesis to operate with a probabilistic model that tracks thousands of universal features to rank candidate patches, not just a limited set of hand-encoded rules. Genesis further differs from these two systems because these two systems operate on manually defined search spaces while Genesis uses automatically inferred search spaces. The inferred search space enables Genesis to operate with productive search spaces that deliver better trade-offs between the coverage and tractability.

DeepFix: DeepFix [45] uses recurrent neural network (RNN) with attention to predict and fix compilation errors in C programs. It is evaluated on a set of student programs collected from an introductory programming class. The techniques in this dissertation differ from DeepFix in both the goal and the program scale. Prophet and Genesis generate correct patches for runtime defects in large open source programs. Directly applying the DeepFix technique to runtime defects would not produce desirable results because the DeepFix model does not consider information from test cases – DeepFix may even generate patches that cannot pass test cases. How to apply deep learning techniques to improve automatic patch generation for runtime defects is still an open question.

9.1.3 Specification-based Techniques

Program Repair with Formal Specifications: Deductive Program Repair formalizes the program repair problem as a program synthesis problem, using the original defective program as a hint [51]. It replaces the expression to repair with a synthesis hole and uses a counterexample-driven synthesis algorithm to find a patch that satisfies the specified pre- and post-conditions. AutoFixE [79] is a program repair tool for

Eiffel programming language. AutoFixE leverages the developer-provided formal specifications (e.g., post-condtions, pre-conditions, and invariants) to automatically find and generate repairs for defects. Cost-aware Program Repair [92] abstracts a C program as a boolean constraint, repairs the constraint based on a cost model, and then concretizes the constraint back to a repaired C program. The goal is to find a repaired program that satisfies all assertions in the program with minimized modification cost. The technique was evaluated on small C programs (less than 50 lines of code) and requires human intervention to define the cost model and to help with the concretization.

Prophet and Genesis differ from these techniques in that they work with large real world applications where formal specifications are typically not available. Note that the learning algorithms in Prophet can apply to these specification-based techniques as well, i.e., if there are multiple patches that satisfy the supplied specifications, the learned model can be used to determine which patch is more likely to be a correct patch.

9.1.4 Targeted Techniques

Researchers have developed a variety of repair and recovery systems that are targeted at specific classes of errors. Comparing to generate-and-validate techniques like Prophet and Genesis, targeted techniques can only be applied to narrow scopes of defects. But they exploit underlying properties of the defects in their targeted scopes to improve the repair or recovery results.

Failure-Oblivous Computing: Failure-oblivious computing [90] checks for out of bounds reads and writes. It discards out of bounds writes and manufactures values for out of bounds reads. This eliminates data corruption from out of bounds writes, eliminates crashes from out of bounds accesses, and enables the program to continue execution along its normal execution path.

Failure-oblivious computing was evaluated on five errors in five server applications. The goal was to enable servers to survive inputs that trigger the errors and continue on to successfully process other inputs. For all five systems, the implemented system realized this goal. For two of the five errors, failure-oblivious computing completely eliminates the error and, on all inputs, delivers the same output as the official developer patch that corrects the error.

Infinite Loop: Bolt [50] attaches to a running application, determines if the application is in an infinite loop, and, if so, exits the loop. A user can also use Bolt to exit a long-running loop. In both cases the goal is to enable the application to continue useful execution. Bolt was evaluated on 13 infinite and 2 long-running loops in 12 applications. For 14 of the 15 loops Bolt delivered a result that was the same or better than terminating the application. For 7 of the 15 loops, Bolt completely eliminates the error and, on all inputs, delivers the same output as the official developer patch that corrects the error. Jolt applies a similar approach but uses the compiler to insert the instrumentation [28]. Infinitel [52] uses SMT solvers to synthesize new loop conditions which terminate the error-triggering executions and leave benign executions intact.

RCV: RCV [61] enables applications to survive null dereference and divide by zero errors. It discards writes via null references, returns zero for reads via null references, and returns zero as the result of divides by zero. Execution continues along the normal execution path.

RCV was evaluated on 18 errors in 7 applications. For 17 of these 18 errors, RCV enables the application to survive the error and continue on successfully process the remaining input. For 11 of the 18 errors, RCV completely eliminates the error and, on all inputs, delivers either identical (9 of 11 errors) or equivalent (2 of 11 errors) outputs as the official developer patch that corrects the error.

DieHard: DieHard [25] provides probabilistic memory safety in the presence of memory errors. In stand-alone mode, DieHard replaces the default memory manager with a memory manager that places objects randomly across a heap to reduce the possibility of memory overwrites due to buffer overflows. In replicated mode, DieHard obtains the final output of the application based on the votes of multiple replications.

Memory Leaks: Cyclic memory allocation eliminates memory leaks by statically bounding the amount of memory that can be allocated at any allocation site [74]. LeakFix [43] proposes to fix memory leaks in C programs by inserting deallocations automatically. LeakFix guarantees that the inserted fix is safe, i.e., the inserted fix will not cause free-before-allocation, double-free, or use-after-free errors.

Integer and Buffer Overflows: TAP automatically discovers and patches integer and buffer overflow errors [95]. TAP uses a template-based approach to generate source-level patches that test for integer or buffer overflows. If an overflow is detected, the patches exit the program before the overflow can occur.

APPEND: APPEND [36] proposes to eliminate null pointer exceptions in Java by applying recovery techniques such as replacing the null pointer with a pointer to an initialized instance of the appropriate class. The presented examples illustrate how this technique can effectively eliminate null pointer exceptions and enhance program survival.

Data Structure Repair: Data structure repair enables applications to recover from data structure corruption errors [33]. Data structure repair enforces a data structure consistency specification. This specification can be provided by a human developer or automatically inferred from correct program executions [34].

Self-Stabilizing Java: Self-Stabilizing Java uses a type system to ensure that the impact of any errors are eventually flushed from the system, returning the system back to a consistent state and promoting successful future execution [38].

9.1.5 Feedback Generation for Programming Education

Singh et al. proposed a technique to automatically generate patches for submitted student Python programs for introductory programming assignments [97]. The technique captures the error model (potential mistakes) of a programming task with a domain specific language. It then takes the submitted Python program and the defined error model as the inputs and produces a sketch [98] that encodes all possible repairs of the submitted program. It finally synthesizes the sketch program to produce a patch with minimal syntactic corrections. Qlose [30] defines program distance based on the number of changes of the behavior of a program with respect to a given set of tests. It attempts find a qualitative program repair that minimizes the program distance between the fixed program and the original program.

Prophet and Genesis differ from those techniques in that Prophet and Genesis generate patches for real-world defects in large applications not programming errors in small student programs. On the other hand, although it would be possible to directly apply Prophet and Genesis to fix student programs, I expect those techniques would outperform Prophet and Genesis on fixing student programs.

The reason is that providing feedbacks for student programs is different from fixing real-world defects in large applications. The submitted student programs for programming assignments are typically small with less than 100 lines of code. It is therefore possible for the sketch tool to fully encode the program logic into SMT formulas. Teachers usually have a reference implementation for each assignment and this reference implementation can be used as a specification for the feedback generation systems.

REFAZER implements an algorithm for learning syntactic program transformations from examples [91]. The REFAZER transformations were used to perform repetitive edits on large code bases and to correct defects in student submissions, and were mostly not useful across assignments. Genesis differs in that it processes patches from multiple applications to derive generalized application-independent transforms that it can apply to fix bugs in yet other (previously unseen) applications. The Genesis transforms also include generators that enable transforms to generate new code (as opposed to simply reusing existing matched code).

9.2 Learning from Existing Programs

Researchers have developed a variety of techniques to learn from existing programs for specification mining, program synthesis, code beautification, and code completion.

Specification Mining and PROSPECTOR: The specification mining technique [23] collects API execution traces of a target library across different applications. The technique then learns a finite state automata that is capable to generate the majority of the collected traces. PROSPECTOR [63] synthesizes a sequence of unary API calls (i.e., jungloids) to derive an object of a specified output type from an object of an input type. It mines API specifications and existing programs with API snippets to construct a graph where each node corresponds to a type and each edge corresponds to an API call. It then computes a path from the node of the input type to the node of the output type to produce an API sequence.

These approaches differ from Prophet and Genesis, because these techniques only focus on API invocations in the program, while Prophet and Genesis reason all kinds of program elements for patch generation. The learning techniques in these approaches rely on deterministic algorithms to produce automata specifications or API sequences, while Prophet and Genesis learn the ranking of generated patches with a probabilistic model.

JSNICE: JSNICE [87] is a JavaScript beautification tool that automatically predicts variable names and generates comments to annotate variable types for JavaScript programs. JSNICE first learns, from a database of JavaScript programs, a probabilistic model of relationships between either pairs of variable names (for predicting variable names) or pairs of types (for generating comments to annotate variable types). Given a new JavaScript program, JSNICE uses a greedy algorithm to search a space of predicted variable names or types, with the learned model guiding the search.

A key difference between JSNICE and Prophet/Genesis is that JSNICE does not aspire to change the program semantics — the goal of JSNICE is instead to change variable names and add comments to beautify the program but leave the original semantics intact. The goal of Prophet and Genesis, in contrast, is to produce correct patches that change the program semantics to eliminate defects. Prophet and Genesis therefore aspire to solve deep semantic problems associated with automatically generating correct program logic. To this end, Prophet and Genesis work with a probabilistic model that combines defect localization information with learned universal properties of correct code. Genesis further works with a novel inference algorithm to infer useful code transforms from human patches.

Code Completion: There is a rich set of work on applying probabilistic model and machine learning learning techniques for code completion [26, 86, 88]. These techniques learn a probabilistic model from a training set of programs and then use the learned model to predict the next token for a partial program. In contrast, our universal property learning algorithm in Prophet and Genesis ranks candidate patches in a search space instead of just the next token. Because the goals are different, our learning technique abstracts away syntactic information like variable names and function names to capture application-independent semantic relationships between the patch and the surrounding code. In contrast, the learning techniques for code completion attempt to cover such syntactic information to precisely predict the next token.

Furthermore, the inference algorithm in Genesis has the high-order goal of inferring transforms that can be applied to a new bug to generate a set of candidate patches. It does not use probabilistic models. It instead obtains candidate transforms with a novel generalization algorithm and formulates the transform selection problem as an integer linear programming.

9.3 Defect Localization

Similar to many other patch generation systems, Prophet and Genesis use defect localization (also called fault localization) techniques to identify potential program locations that are relevant to a software defect. They then modify those identified locations to generate candidate patches.

Spectrum-based Localization: Spectrum-based defect localization techniques [27, 47, 72, 105] collect the execution traces of test cases on the program. The techniques then track the frequencies of each program statement being executed in positive test cases and negative test cases. Most of the techniques use the tracked frequencies to compute a suspicious score for each statement and rank all statements by the order of their suspicious scores. The defect localization algorithm in Prophet is also spectrum-based. There are many different formulas proposed to compute the suspicious score based on the tracked frequencies [27, 47, 72, 105], but recently Pearson et al. show that the performance difference of different formulas is statistically insignificant on real-world defects [78].

Mutation-Based Localization: Mutation-based defect localization techniques [70, 75, 76] extend spectrum-based techniques to consider the importance of an executed statement. The techniques mutate each statement in the program and check whether mutating the statement changes the results of negative test cases and positive test cases. The more often the statement changes the results of negative cases and less often changes the results of positive test cases, the more suspicious the statement is considered.

In fact, Kali described in Section 2.8 could be used as a mutation-based defect localization system as well. Our results indicate that the generated functionality elimination patches can often help developers to pinpoint the root cause of a defect. Indeed, if removing a statement or a branch can cause the defective program to pass all test cases, the removed part is very likely to be the root cause of the defect.

9.4 Code Refactoring

SYDIT [66] and Lase [67] extract edit scripts from one (SYDIT) or more (Lase) example edits. The script is a sequential list of modification operations that insert statements or update existing statements. SYDIT and Lase then generate changes to other code snippets in the same application with the goal of automating repetitive edits. RASE [68] uses Lase edit scripts to refactor code clones. FixMeUp [100] works with access control templates that implement policies for sensitive operations. Using these templates, FixMeUp finds unprotected sensitive operations and inserts appropriate checks. An analysis of the application can extract an application-specific template [99], which FixMeUp can then apply across the same application. The inference algorithm in Genesis differs in that it processes multiple patches from multiple applications to derive generalized application-independent transforms that it can apply to fix bugs in yet other applications. The Genesis transforms include generators so that transforms can generate multiple candidate patches (as opposed to a single edit as in SYDIT, Lase, and FixMeUp).

9.5 Discussion

In general, Prophet and Genesis differ from many previous generate-and-validate systems because these previous systems only rely on the information from the supplied test cases with the goal of finding plausible (but not necessarily correct) patches in a hand-coded search space. Prophet and Genesis automatically learn universal properties and patching strategies of past successful patches to recognize and prioritize correct patches among multiple plausible patches in an automatically inferred search space. To the best of my knowledge, Prophet is the first generate-and-validate system that learns a probabilistic model of correct patches. Genesis is the first generate-and-validate system that automatically infers code transforms and search spaces from human patches.

Chapter 10

Conclusion

Automatic patch generation holds out the promise of automatically correcting software defects without human interventions. Standard generate-and-validate systems formulate the patch generation as a search problem. Therefore such a system operates in a completely different way than human developers. The system exhaustively enumerates all candidate patches in its search space with its superior computation power, while human developers often unconsciously apply certain software engineering patching strategies to consider only those patches that exhibit certain properties of successful code.

This dissertation presents a new automatic patch generation approach that combines the superior computation power of machines with the sophisticated insights of human developers. Specifically, this dissertation presents novel learning and inference algorithms that extract universal patching strategies and universal properties from past successful human patches. This dissertation also presents new generate-and-validate frameworks that integrate the learned and inferred information into the patch generation process. Powered by the automatically extracted human knowledge, our two prototype systems, Prophet and Genesis, generate correct patches for significantly more benchmark defects than previous generate-and-validate systems.

The experimental results in this dissertation are consistent with our hypotheses. Correct patches across different applications share universal properties and patching strategies that can be automatically learned from past successful human patches.

Our results demonstrate that the universal patching strategies and properties of successful human patches, if appropriately extracted and integrated into an automatic patch generation system, will significantly improve the capability of the system to generate correct patches for new defects. Exploiting information of human patches is a promising direction for automatic patch generation. The work presented in this dissertation lays the foundation for future progress in this direction.

My work also demonstrates that the growing volume of programs is not just a challenge but also a great opportunity. This opportunity enables new learning and inference techniques such as Prophet and Genesis that were impossible before. Besides patch generation, I believe the ideas of the presented learning and inference techniques in this dissertation can be applied to other software engineering tasks such as program synthesis and code refactoring. As the number of software programs continues to grow, I expect automated programming techniques that exploit information of existing programs will become more powerful in future.

Appendix A

Prophet and SPR Results Per Search Space Configuration

Tables A.1-A.32 present the detailed experimental results. Each table presents the results of SPR or Prophet on one search space configuration. The first column of the table presents the defect id. The second column of the table presents the total number of candidate patch templates in the search space. The third column presents the number of patch templates that manipulate branch conditions. The fourth column presents the total number of evaluated patch templates in 12 hours. The fifth column presents the total number of evaluated condition patch templates during in 12 hours. The sixth column presents the number of templates for which generate plausible patches. The seventh column presents the number of condition templates for which generate plausible patches. The eighth column presents the total number of plausible patches the system finds in 12 hours. The ninth column presents the number of plausible patches which manipulate branch conditions. The tenth column presents the number of correct patches the system finds in 12 hours. The eleventh column presents the rank of the template that generates the first correct patch in the search space. The twelfth column presents the rank of the template among plausible templates. The last column presents the rank of the correct patch among all generated plausible patches.

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	10949	3394	10949	3394	33	28	111	106	0	-	-	-
libtiff-5b0217-3dfb33	83937	19298	52303	16953	208	147	211	150	0	-	-	-
libtiff-d13be7-ccadf4	184068	141307	63046	58498	1423	1423	1703	1423	1	1093	1	1
lighttpd-2661-2662	34688	27541	34688	27541	70	8	73	11	0	-	-	-
lighttpd-1913-1914	32903	21707	32903	21707	4	4	4	4	0	-	-	-
python-69934-69935	17828	5874	6501	4383	0	0	0	0	0	-	-	-
gmp-13420-13421	29703	5319	29703	5319	0	0	0	0	0	-	-	-
gzip-a1d3d4-f17cbd	25303	7535	25303	7535	5	0	5	0	1	1135	1	1
python-70056-70059	22537	9353	4106	3644	0	0	0	0	0	-	-	-
fbc-5458-5459	4864	2144	4864	2144	26	26	46	35	2	19	1	1
libtiff-ee2ce5-b5691a	34863	25186	34863	25186	328	328	328	328	1	110	1	1
php-310991-310999	47688	11762	21206	9962	1	1	1	1	1	808	1	1
php-308734-308761	14	11	14	11	0	0	0	0	0	-	-	-
php-308262-308315	22003	1869	5283	1869	0	0	0	0	0	150	-	-
php-307562-307561	21252	4869	13996	4869	1	0	1	0	1	2221	1	1
php-309579-309580	41568	6890	19370	6890	2	2	2	2	1	288	1	1
php-310011-310050	34470	8642	2394	2082	63	13	69	23	1	640	13	21
php-309688-309716	47600	11980	2609	2540	71	71	95	95	1	2740	64	86
php-309516-309535	19142	4578	19142	4578	1	0	1	0	1	8851	1	1
php-307846-307853	13971	3009	13971	3009	1	0	1	0	1	8470	1	1
php-311346-311348	8096	3027	8096	3027	50	38	72	60	2	25	1	1
php-307914-307915	25707	7731	25707	7731	1	0	1	0	1	1	1	1
php-309111-309159	32541	6583	32541	6583	10	1	10	1	1	6838	9	9
php-309892-309910	8665	1432	8665	1432	21	17	26	22	4	161	1	1

Table A.1: Prophet-100 Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	10949	3394	10949	3394	29	24	56	51	0	-	-	-
libtiff-5b0217-3dfb33	83937	19298	1249	1112	146	146	3192	3192	0	-	-	-
libtiff-d13be7-ccadf4	184068	141307	3938	3896	116	116	4540	4362	2	1093	1	1
lighttpd-2661-2662	34688	27541	34688	27541	78	16	109	47	0	-	-	-
lighttpd-1913-1914	32903	21707	32903	21707	4	4	12	12	0	-	-	-
python-69934-69935	17828	5874	6501	4383	0	0	0	0	0	-	-	-
gmp-13420-13421	29703	5319	29703	5319	0	0	0	0	0	-	-	-
gzip-a1d3d4-f17cbd	25303	7535	25303	7535	5	0	5	0	1	1135	1	1
python-70056-70059	22537	9353	4106	3644	0	0	0	0	0	-	-	-
fbc-5458-5459	4864	2144	4864	2144	23	23	98	71	0	19	-	-
libtiff-ee2ce5-b5691a	34863	25186	4576	4282	113	113	3889	3889	1	110	1	1
php-310991-310999	47688	11762	20995	9962	1	1	1	1	1	808	1	1
php-308734-308761	14	11	14	11	0	0	0	0	0	-	-	-
php-308262-308315	22003	1869	2019	1510	18	18	44	44	1	150	2	2
php-307562-307561	21252	4869	13736	4869	1	0	1	0	1	2221	1	1
php-309579-309580	41568	6890	15676	6890	11	11	38	38	1	288	6	12
php-310011-310050	34470	8642	2424	2082	71	7	76	14	1	640	7	9
php-309688-309716	47600	11980	538	501	2	2	96	96	0	2740	-	-
php-309516-309535	19142	4578	19142	4578	1	0	1	0	1	8851	1	1
php-307846-307853	13971	3009	13971	3009	1	0	1	0	1	8470	1	1
php-311346-311348	8096	3027	1485	1397	5	5	101	101	1	25	1	1
php-307914-307915	25707	7731	25707	7731	1	0	1	0	1	1	1	1
php-309111-309159	32541	6583	32541	6583	10	1	10	1	1	6838	9	9
php-309892-309910	8665	1432	451	345	11	11	96	96	1	161	2	12

Table A.2: Prophet-100-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	12718	3394	12718	3394	33	28	111	106	0	-	-	-
libtiff-5b0217-3dfb33	88613	19298	53568	16729	210	147	213	150	0	-	-	-
libtiff-d13be7-ccadf4	186777	141307	60747	54316	1185	1185	1405	1185	1	1202	1	1
lighttpd-2661-2662	35810	27541	35810	27541	66	4	69	7	0	-	-	-
lighttpd-1913-1914	34917	21707	34917	21707	4	4	4	4	0	-	-	-
python-69934-69935	18168	5874	6408	4339	0	0	0	0	0	-	-	-
gmp-13420-13421	49743	5319	49743	5319	0	0	0	0	0	-	-	-
gzip-a1d3d4-f17cbd	31251	7535	31251	7535	5	0	5	0	1	715	1	1
python-70056-70059	26398	9353	3950	3517	0	0	0	0	0	-	-	-
fbc-5458-5459	5938	2144	5938	2144	28	26	48	35	2	16	1	1
libtiff-ee2ce5-b5691a	37661	25186	37661	25186	328	328	328	328	1	149	1	1
php-310991-310999	48843	11762	19316	8758	1	1	1	1	1	1509	1	1
php-308734-308761	14	11	14	11	0	0	0	0	0	-	-	-
php-308262-308315	22472	1869	2963	1648	4	3	6	5	1	206	1	1
php-307562-307561	21944	4869	14071	4869	1	0	1	0	1	2962	1	1
php-309579-309580	42322	6890	19229	6890	2	2	2	2	1	330	1	1
php-310011-310050	35256	8642	4112	3411	60	13	70	23	1	742	9	14
php-309688-309716	48118	11980	3019	2899	68	68	92	92	0	6827	-	-
php-309516-309535	21008	4578	21008	4578	1	0	1	0	1	9788	1	1
php-307846-307853	16435	3009	16435	3009	1	0	1	0	1	10250	1	1
php-311346-311348	8173	3027	8173	3027	51	38	73	60	2	17	1	1
php-307914-307915	27121	7731	27121	7731	1	0	1	0	1	1	1	1
php-309111-309159	34080	6583	34080	6583	10	1	10	1	1	8261	10	10
php-309892-309910	10855	1432	3184	1368	64	17	69	22	4	313	1	1

Table A.3: Prophet-100-RExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	12718	3394	12718	3394	29	24	56	51	0	-	-	-
libtiff-5b0217-3dfb33	88613	19298	1455	1325	146	146	3192	3192	0	-	-	-
libtiff-d13be7-ccadf4	186777	141307	4252	4207	115	115	4525	4347	2	1202	1	1
lighttpd-2661-2662	35810	27541	35810	27541	74	12	97	35	0	-	-	-
lighttpd-1913-1914	34917	21707	34917	21707	4	4	12	12	0	-	-	-
python-69934-69935	18168	5874	6402	4339	0	0	0	0	0	-	-	-
gmp-13420-13421	49743	5319	49743	5319	0	0	0	0	0	-	-	-
gzip-a1d3d4-f17cbd	31251	7535	31251	7535	5	0	5	0	1	715	1	1
python-70056-70059	26398	9353	3967	3517	0	0	0	0	0	-	-	-
fbc-5458-5459	5938	2144	5938	2144	25	23	100	71	0	16	-	-
libtiff-ee2ce5-b5691a	37661	25186	5031	4747	113	113	3608	3608	1	149	1	1
php-310991-310999	48843	11762	20496	9251	1	1	1	1	1	1509	1	1
php-308734-308761	14	11	14	11	0	0	0	0	0	-	-	-
php-308262-308315	22472	1869	2163	1526	18	18	43	43	1	206	2	2
php-307562-307561	21944	4869	14132	4869	1	0	1	0	1	2962	1	1
php-309579-309580	42322	6890	15022	6890	11	11	38	38	1	330	6	12
php-310011-310050	35256	8642	4122	3411	58	7	64	14	1	742	7	9
php-309688-309716	48118	11980	920	857	2	2	92	92	0	6827	-	-
php-309516-309535	21008	4578	21008	4578	1	0	1	0	1	9788	1	1
php-307846-307853	16435	3009	16435	3009	1	0	1	0	1	10250	1	1
php-311346-311348	8173	3027	1462	1363	5	5	106	106	1	17	1	1
php-307914-307915	27121	7731	27121	7731	1	0	1	0	1	1	1	1
php-309111-309159	34080	6583	34080	6583	10	1	10	1	1	8261	10	10
php-309892-309910	10855	1432	470	372	11	11	100	100	1	313	2	12

Table A.4: Prophet-100-RExt-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	in Plausible
gmp-14166-14167	39287	14161	39287	14161	34	28	112	106	0	-	-	-
libtiff-5b0217-3dfb33	221134	65017	64363	31834	247	149	250	152	0	50770	-	-
libtiff-d13be7-ccadf4	296426	228659	70050	65731	1423	1423	1703	1423	1	1183	1	1
lighttpd-2661-2662	120263	98028	79882	67844	48	2	50	4	0	-	-	-
lighttpd-1913-1914	68199	48133	37214	30701	58	58	58	58	0	-	-	-
python-69934-69935	47544	13775	6546	4750	0	0	0	0	0	-	-	-
gmp-13420-13421	50672	8763	50672	8763	3	0	3	0	2	14102	1	1
gzip-a1d3d4-f17cbd	48702	15104	48702	15104	14	0	14	0	1	1929	1	1
python-70056-70059	39599	17961	4032	3645	0	0	0	0	0	-	-	-
fbc-5458-5459	9857	4495	9857	4495	37	37	61	46	2	33	1	1
libtiff-ee2ce5-b5691a	171379	106867	137262	103332	328	328	328	328	1	280	1	1
php-310991-310999	89230	18988	21250	11637	1	1	1	1	1	907	1	1
php-308734-308761	14692	4160	14692	4160	4	4	4	4	2	5376	1	1
php-308262-308315	90431	10845	8496	7508	3	3	5	5	1	1365	1	1
php-307562-307561	31597	6997	14698	6997	1	0	1	0	1	2672	1	1
php-309579-309580	60351	11416	23605	11416	2	2	2	2	1	767	1	1
php-310011-310050	77671	16558	4857	4556	63	13	69	23	1	1348	13	21
php-309688-309716	71633	15744	3310	3241	68	68	92	92	1	3465	61	83
php-309516-309535	27098	6314	27098	6314	1	0	1	0	1	10954	1	1
php-307846-307853	22131	4757	16871	4709	1	0	1	0	1	10742	1	1
php-311346-311348	9799	3879	9799	3879	50	38	72	60	2	27	1	1
php-307914-307915	47988	15066	35684	15066	1	0	1	0	1	1	1	1
php-309111-309159	52908	12232	36533	11928	10	1	10	1	1	7701	9	9
php-309892-309910	40758	9999	13614	7118	21	17	26	22	4	462	1	1

Table A.5: Prophet-200 Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	39287	14161	39287	14161	30	24	57	51	0	-	-	-
libtiff-5b0217-3dfb33	221134	65017	2054	1940	146	146	2776	2776	0	50770	-	-
libtiff-d13be7-ccadf4	296426	228659	4679	4657	111	111	4308	4191	2	1183	1	1
lighttpd-2661-2662	120263	98028	77400	65714	55	10	75	30	0	-	-	-
lighttpd-1913-1914	68199	48133	22474	19567	56	56	160	160	0	-	-	-
python-69934-69935	47544	13775	6616	4760	0	0	0	0	0	-	-	-
gmp-13420-13421	50672	8763	50672	8763	3	0	3	0	2	14102	1	1
gzip-a1d3d4-f17cbd	48702	15104	48702	15104	14	0	14	0	1	1929	1	1
python-70056-70059	39599	17961	4464	4074	0	0	0	0	0	-	-	-
fbc-5458-5459	9857	4495	6132	3920	34	34	118	89	0	33	-	-
libtiff-ee2ce5-b5691a	171379	106867	8258	7714	113	113	3458	3458	1	280	1	1
php-310991-310999	89230	18988	21436	11684	1	1	1	1	1	907	1	1
php-308734-308761	14692	4160	14692	4160	4	4	4	4	2	5376	1	1
php-308262-308315	90431	10845	7641	7304	15	15	36	36	1	1365	2	2
php-307562-307561	31597	6997	15277	6997	1	0	1	0	1	2672	1	1
php-309579-309580	60351	11416	18816	11416	11	11	38	38	1	767	6	11
php-310011-310050	77671	16558	4867	4556	61	7	69	14	1	1348	7	9
php-309688-309716	71633	15744	538	501	2	2	93	93	0	3465	-	-
php-309516-309535	27098	6314	27098	6314	1	0	1	0	1	10954	1	1
php-307846-307853	22131	4757	16927	4709	1	0	1	0	1	10742	1	1
php-311346-311348	9799	3879	1799	1719	5	5	103	103	1	27	1	1
php-307914-307915	47988	15066	35979	15066	1	0	1	0	1	1	1	1
php-309111-309159	52908	12232	36614	11949	10	1	10	1	1	7701	9	9
php-309892-309910	40758	9999	1417	1316	11	11	89	89	1	462	2	10

Table A.6: Prophet-200-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	61171	14161	53629	13793	33	28	111	106	0	-	-	-
libtiff-5b0217-3dfb33	278857	65017	52657	22730	377	149	380	152	0	70740	-	-
libtiff-d13be7-ccadf4	300745	228659	70497	63153	819	819	939	819	1	1321	1	1
lighttpd-2661-2662	124449	98028	78470	64364	62	0	62	0	0	-	-	-
lighttpd-1913-1914	70712	48133	35590	27952	54	54	54	54	0	-	-	-
python-69934-69935	48560	13775	6699	4826	0	0	0	0	0	24090	-	-
gmp-13420-13421	79763	8763	64198	8713	1	0	1	0	1	39181	1	1
gzip-a1d3d4-f17cbd	58432	15104	58432	15104	14	0	14	0	1	1303	1	1
python-70056-70059	43598	17961	4428	4021	0	0	0	0	0	-	-	-
fbc-5458-5459	11828	4495	9775	4353	30	28	50	37	2	24	1	1
libtiff-ee2ce5-b5691a	190629	106867	119616	81875	328	328	328	328	1	408	1	1
php-310991-310999	90936	18988	21023	11071	1	1	1	1	1	1778	1	1
php-308734-308761	18784	4160	18784	4160	4	4	4	4	2	6242	1	1
php-308262-308315	92459	10845	8606	7477	4	3	6	5	1	1817	1	1
php-307562-307561	32546	6997	15557	6997	1	0	1	0	1	3481	1	1
php-309579-309580	61407	11416	22933	11416	2	2	2	2	1	603	1	1
php-310011-310050	78981	16558	6593	5934	52	24	66	41	1	1413	9	14
php-309688-309716	73424	15744	2177	2081	68	68	92	92	0	8488	-	-
php-309516-309535	29514	6314	29514	6314	1	0	1	0	1	11765	1	1
php-307846-307853	25447	4757	18938	4661	1	0	1	0	1	12791	1	1
php-311346-311348	9978	3879	10051	3879	52	38	74	60	2	20	1	1
php-307914-307915	50442	15066	36291	15066	1	0	1	0	1	1	1	1
php-309111-309159	58903	12232	34193	11196	10	1	10	1	1	9220	10	10
php-309892-309910	52975	9999	7971	4457	35	17	40	22	4	938	1	1

Table A.7: Prophet-200-RExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	61171	14161	55865	14057	29	24	56	51	0	-	-	-
libtiff-5b0217-3dfb33	278857	65017	2032	1923	146	146	2806	2806	0	70740	-	-
libtiff-d13be7-ccadf4	300745	228659	4825	4803	109	109	4279	4162	2	1321	1	1
lighttpd-2661-2662	124449	98028	77686	64125	64	10	81	27	0	-	-	-
lighttpd-1913-1914	70712	48133	23341	19172	55	55	155	155	0	-	-	-
python-69934-69935	48560	13775	6699	4826	0	0	0	0	0	24090	-	-
gmp-13420-13421	79763	8763	63696	8713	1	0	1	0	1	39181	1	1
gzip-a1d3d4-f17cbd	58432	15104	58432	15104	14	0	14	0	1	1303	1	1
python-70056-70059	43598	17961	4385	3978	0	0	0	0	0	-	-	-
fbc-5458-5459	11828	4495	8182	4060	29	27	107	78	0	24	-	-
libtiff-ee2ce5-b5691a	190629	106867	6895	6520	113	113	3429	3429	1	408	1	1
php-310991-310999	90936	18988	20884	11014	1	1	1	1	1	1778	1	1
php-308734-308761	18784	4160	18784	4160	4	4	4	4	2	6242	1	1
php-308262-308315	92459	10845	7422	7082	15	15	36	36	1	1817	2	2
php-307562-307561	32546	6997	15557	6997	1	0	1	0	1	3481	1	1
php-309579-309580	61407	11416	17325	11416	11	11	38	38	1	603	6	11
php-310011-310050	78981	16558	6623	5934	54	12	61	19	1	1413	7	9
php-309688-309716	73424	15744	920	857	2	2	91	91	0	8488	-	-
php-309516-309535	29514	6314	29514	6314	1	0	1	0	1	11765	1	1
php-307846-307853	25447	4757	18964	4661	1	0	1	0	1	12791	1	1
php-311346-311348	9978	3879	1733	1643	5	5	105	105	1	20	1	1
php-307914-307915	50442	15066	36465	15066	1	0	1	0	1	1	1	1
php-309111-309159	58903	12232	33503	11036	10	1	10	1	1	9220	10	10
php-309892-309910	52975	9999	1413	1305	11	11	84	84	1	938	2	10

Table A.8: Prophet-200-RExt-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	59327	22055	49749	21743	33	28	111	106	0	-	-	-
libtiff-5b0217-3dfb33	276142	85251	64841	32860	225	149	228	152	0	52950	-	-
libtiff-d13be7-ccadf4	568172	432083	76111	71854	1423	1423	1703	1423	1	1453	1	1
lighttpd-2661-2662	191579	159862	92356	80355	48	3	51	6	0	-	-	-
lighttpd-1913-1914	159739	114667	36479	30861	63	63	63	63	0	-	-	-
python-69934-69935	65326	18300	7258	5562	0	0	0	0	0	-	-	-
gmp-13420-13421	69661	12794	61424	12794	9	5	9	5	2	14989	6	6
gzip-a1d3d4-f17cbd	82349	18024	82349	18024	14	0	14	0	1	2250	1	1
python-70056-70059	58530	26327	4453	4155	0	0	0	0	0	-	-	-
fbc-5458-5459	16353	5416	10124	5416	28	28	48	37	2	33	1	1
libtiff-ee2ce5-b5691a	218252	142593	151378	118052	328	328	328	328	1	308	1	1
php-310991-310999	139311	31061	27055	18568	1	1	1	1	1	1396	1	1
php-308734-308761	30623	7437	13609	7432	4	4	4	4	2	7684	1	1
php-308262-308315	150963	19358	12615	11830	2	2	4	4	1	2046	1	1
php-307562-307561	60507	15748	18133	10420	1	0	1	0	1	4780	1	1
php-309579-309580	101940	17784	24749	17625	2	2	2	2	1	875	1	1
php-310011-310050	105542	23824	7441	7000	55	14	64	25	1	2203	14	23
php-309688-309716	95206	19747	528	501	68	68	90	90	1	3758	61	82
php-309516-309535	52377	12093	35403	11940	1	0	1	0	1	12002	1	1
php-307846-307853	40272	8051	16704	6229	3	2	4	3	1	12101	1	1
php-311346-311348	14543	5619	8465	5526	50	41	72	63	2	28	1	1
php-307914-307915	64378	20107	36206	18364	1	0	1	0	1	1	1	1
php-309111-309159	86627	22335	34642	13196	10	1	10	1	1	8749	9	9
php-309892-309910	62347	19484	16588	10223	21	17	26	22	4	532	1	1

Table A.9: Prophet-300 Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	59327	22055	51701	21934	29	24	56	51	0	-	-	-
libtiff-5b0217-3dfb33	276142	85251	2433	2326	146	146	2794	2794	0	52950	-	-
libtiff-d13be7-ccadf4	568172	432083	10788	10704	108	108	4237	4120	2	1453	1	1
lighttpd-2661-2662	191579	159862	90638	78887	59	14	80	35	0	-	-	-
lighttpd-1913-1914	159739	114667	22555	20308	57	57	160	160	0	42256	-	-
python-69934-69935	65326	18300	7307	5579	0	0	0	0	0	-	-	-
gmp-13420-13421	69661	12794	60776	12794	7	3	22	18	2	14989	4	19
gzip-a1d3d4-f17cbd	82349	18024	82349	18024	14	0	14	0	1	2250	1	1
python-70056-70059	58530	26327	4507	4200	0	0	0	0	0	1495	-	-
fbc-5458-5459	16353	5416	6663	4350	34	34	118	89	0	33	-	-
libtiff-ee2ce5-b5691a	218252	142593	9848	9310	113	113	3467	3467	1	308	1	1
php-310991-310999	139311	31061	27584	18695	1	1	1	1	1	1396	1	1
php-308734-308761	30623	7437	13646	7432	4	4	4	4	2	7684	1	1
php-308262-308315	150963	19358	12072	11728	10	10	26	26	1	2046	2	2
php-307562-307561	60507	15748	18148	10436	1	0	1	0	1	4780	1	1
php-309579-309580	101940	17784	18107	14178	11	11	38	38	1	875	6	12
php-310011-310050	105542	23824	7451	7000	57	7	61	14	1	2203	7	9
php-309688-309716	95206	19747	528	501	2	2	90	90	0	3758	-	-
php-309516-309535	52377	12093	35698	11940	1	0	1	0	1	12002	1	1
php-307846-307853	40272	8051	15952	6206	5	4	12	11	1	12101	1	1
php-311346-311348	14543	5619	1923	1843	8	8	99	99	1	28	1	1
php-307914-307915	64378	20107	36360	18402	1	0	1	0	1	1	1	1
php-309111-309159	86627	22335	35676	13501	10	1	10	1	1	8749	9	9
php-309892-309910	62347	19484	1925	1845	11	11	81	81	1	532	2	10

Table A.10: Prophet-300-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	95751	22055	43517	13623	31	28	109	106	0	62583	-	-
libtiff-5b0217-3dfb33	336808	85251	53883	23725	303	149	306	152	0	72938	-	-
libtiff-d13be7-ccadf4	573714	432083	76021	68438	556	556	609	556	1	1608	1	1
lighttpd-2661-2662	197676	159862	90518	76321	58	4	61	7	0	-	-	-
lighttpd-1913-1914	168946	114667	36761	29588	60	60	60	60	0	-	-	-
python-69934-69935	66616	18300	7338	5570	0	0	0	0	0	26404	-	-
gmp-13420-13421	107650	12794	65759	12094	9	5	9	5	1	40564	6	6
gzip-a1d3d4-f17cbd	100652	18024	102416	18024	14	0	14	0	1	1588	1	1
python-70056-70059	64192	26327	4479	4184	0	0	0	0	0	-	-	-
fbc-5458-5459	18782	5416	9032	4753	32	30	52	39	2	24	1	1
libtiff-ee2ce5-b5691a	241707	142593	121791	85492	328	328	328	328	1	444	1	1
php-310991-310999	142164	31061	26773	17940	1	1	1	1	1	2939	1	1
php-308734-308761	36012	7437	14040	6893	4	4	4	4	2	8317	1	1
php-308262-308315	159696	19358	12721	11808	2	2	4	4	1	2694	1	1
php-307562-307561	64392	15748	17926	10160	1	0	1	0	1	5763	1	1
php-309579-309580	103151	17784	24041	16881	2	2	2	2	1	749	1	1
php-310011-310050	107479	23824	9297	8378	44	25	62	43	1	2233	10	16
php-309688-309716	97372	19747	900	857	68	68	88	88	0	9296	-	-
php-309516-309535	56549	12093	35228	11710	1	0	1	0	1	12585	1	1
php-307846-307853	44992	8051	19506	6176	3	2	4	3	1	14098	1	1
php-311346-311348	14889	5619	8610	5526	50	41	72	63	2	21	1	1
php-307914-307915	67286	20107	36344	17795	1	0	1	0	1	1	1	1
php-309111-309159	102384	22335	32495	12321	10	1	10	1	1	10350	10	10
php-309892-309910	74780	19484	11247	7175	28	17	33	22	4	1194	1	1

Table A.11: Prophet-300-RExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	95751	22055	45224	14092	27	24	54	51	0	62583	-	-
libtiff-5b0217-3dfb33	336808	85251	2369	2261	146	146	2848	2848	0	72938	-	-
libtiff-d13be7-ccadf4	573714	432083	11655	11585	107	107	4205	4088	2	1608	1	1
lighttpd-2661-2662	197676	159862	90748	76551	57	3	59	5	0	-	-	-
lighttpd-1913-1914	168946	114667	23858	20291	54	54	149	149	0	46828	-	-
python-69934-69935	66616	18300	7948	5824	0	0	0	0	0	26404	-	-
gmp-13420-13421	107650	12794	65438	12094	4	3	19	18	1	40564	4	19
gzip-a1d3d4-f17cbd	100652	18024	102416	18024	14	0	14	0	1	1588	1	1
python-70056-70059	64192	26327	4479	4184	0	0	0	0	0	1570	-	-
fbc-5458-5459	18782	5416	7041	4022	33	31	113	84	0	24	-	-
libtiff-ee2ce5-b5691a	241707	142593	8496	8118	113	113	3425	3425	1	444	1	1
php-310991-310999	142164	31061	26306	17900	1	1	1	1	1	2939	1	1
php-308734-308761	36012	7437	13969	6893	4	4	4	4	2	8317	1	1
php-308262-308315	159696	19358	11843	11506	10	10	25	25	1	2694	2	2
php-307562-307561	64392	15748	17976	10160	1	0	1	0	1	5763	1	1
php-309579-309580	103151	17784	17004	12979	11	11	38	38	1	749	6	12
php-310011-310050	107479	23824	9327	8378	50	12	60	19	1	2233	7	9
php-309688-309716	97372	19747	900	857	2	2	88	88	0	9296	-	-
php-309516-309535	56549	12093	34931	11710	1	0	1	0	1	12585	1	1
php-307846-307853	44992	8051	19371	6176	4	3	5	4	1	14098	1	1
php-311346-311348	14889	5619	1883	1795	8	8	102	102	1	21	1	1
php-307914-307915	67286	20107	36699	17978	1	0	1	0	1	1	1	1
php-309111-309159	102384	22335	32398	12291	10	1	10	1	1	10350	10	10
php-309892-309910	74780	19484	1925	1845	11	11	86	86	1	1194	2	10

Table A.12: Prophet-300-RExt-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	448318	141182	43569	22779	31	28	109	106	0	-	-	-
libtiff-5b0217-3dfb33	1941177	1352092	83094	59078	182	149	185	152	0	91329	-	-
libtiff-d13be7-ccadf4	2353240	1771135	113514	106810	416	416	431	416	1	2820	1	1
lighttpd-2661-2662	1269307	1016413	120469	107039	46	10	54	18	0	1197010	-	-
lighttpd-1913-1914	1183286	935121	59987	56385	51	51	51	51	0	-	-	-
python-69934-69935	863452	386935	39078	29758	0	0	0	0	0	-	-	-
gmp-13420-13421	389124	104714	60147	20342	8	5	8	5	1	22027	6	6
gzip-a1d3d4-f17cbd	341681	92972	137970	43616	16	0	16	0	2	4299	1	1
python-70056-70059	526778	236707	4555	4551	0	0	0	0	0	-	-	-
fbc-5458-5459	16353	5416	8882	5169	28	28	47	36	2	33	1	1
libtiff-ee2ce5-b5691a	3237374	2609366	145761	129703	328	328	328	328	1	2601	1	1
php-310991-310999	783770	185001	56088	55384	1	1	1	1	1	8294	1	1
php-308734-308761	440926	151934	43304	38805	0	0	0	0	0	45726	-	-
php-308262-308315	637558	192659	57001	56034	1	1	2	2	1	8304	1	1
php-307562-307561	637588	163053	59252	53402	1	0	1	0	1	34355	1	1
php-309579-309580	621880	198376	65289	59970	2	2	2	2	1	7010	1	1
php-310011-310050	582198	162849	653	653	0	0	0	0	0	13331	-	-
php-309688-309716	609953	191952	30720	30659	34	34	35	35	0	34218	-	-
php-309516-309535	496992	136030	33781	27530	0	0	0	0	0	41978	-	-
php-307846-307853	477171	130366	32604	26252	0	0	0	0	0	43891	-	-
php-311346-311348	1030763	123671	6068	6068	38	38	45	45	2	303	1	1
php-307914-307915	539078	175191	55905	49061	1	0	1	0	1	1	1	1
php-309111-309159	513188	178631	56649	49491	8	1	8	1	0	31831	-	-
php-309892-309910	498669	177309	48111	46807	17	17	22	22	3	5300	1	1

Table A.13: Prophet-2000 Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	448318	141182	45987	23728	27	24	54	51	0	-	-	-
libtiff-5b0217-3dfb33	1941177	1352092	21932	21903	111	111	2276	2276	0	91329	-	-
libtiff-d13be7-ccadf4	2353240	1771135	34658	34608	103	103	3559	3446	2	2820	1	1
lighttpd-2661-2662	1269307	1016413	115409	103056	57	23	92	58	0	1197010	-	-
lighttpd-1913-1914	1183286	935121	57046	54412	40	40	108	108	0	81194	-	-
python-69934-69935	863452	386935	34011	28504	0	0	0	0	0	-	-	-
gmp-13420-13421	389124	104714	58786	20230	6	3	21	18	1	22027	4	19
gzip-a1d3d4-f17cbd	341681	92972	137153	43377	16	0	16	0	2	4299	1	1
python-70056-70059	526778	236707	5424	5420	0	0	0	0	0	7689	-	-
fbc-5458-5459	16353	5416	6663	4350	34	34	118	89	0	33	-	-
libtiff-ee2ce5-b5691a	3237374	2609366	53567	53119	101	101	1964	1964	1	2601	1	1
php-310991-310999	783770	185001	56048	55384	1	1	1	1	1	8294	1	1
php-308734-308761	440926	151934	43302	38805	0	0	0	0	0	45726	-	-
php-308262-308315	637558	192659	15637	15631	9	9	19	19	1	8304	2	2
php-307562-307561	637588	163053	60720	53760	1	0	1	0	1	34355	1	1
php-309579-309580	621880	198376	33743	33664	11	11	38	38	1	7010	6	11
php-310011-310050	582198	162849	653	653	0	0	0	0	0	13331	-	-
php-309688-309716	609953	191952	7138	7121	13	13	40	40	0	34218	-	-
php-309516-309535	496992	136030	34112	27593	0	0	0	0	0	41978	-	-
php-307846-307853	477171	130366	32634	26252	0	0	0	0	0	43891	-	-
php-311346-311348	1030763	123671	6068	6068	3	3	45	45	1	303	1	1
php-307914-307915	539078	175191	56054	49117	1	0	1	0	1	1	1	1
php-309111-309159	513188	178631	56630	49491	8	1	8	1	0	31831	-	-
php-309892-309910	498669	177309	33432	33262	0	0	0	0	0	5300	-	-

Table A.14: Prophet-2000-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	780425	141182	35880	14864	30	28	108	106	0	87034	-	-
libtiff-5b0217-3dfb33	2027218	1352092	71298	51369	154	149	157	152	0	112099	-	-
libtiff-d13be7-ccadf4	2414317	1771135	113033	104624	416	416	431	416	1	4324	1	1
lighttpd-2661-2662	1290186	1016413	110266	95506	60	11	68	19	0	1217473	-	-
lighttpd-1913-1914	1206699	935121	60268	55919	48	48	48	48	0	-	-	-
python-69934-69935	937134	386935	34097	28463	0	0	0	0	0	55996	-	-
gmp-13420-13421	823835	104714	61757	17041	6	5	6	5	1	48675	6	6
gzip-a1d3d4-f17cbd	399573	92972	127371	34577	14	0	14	0	1	6336	2	2
python-70056-70059	573639	236707	5114	5110	0	0	0	0	0	-	-	-
fbc-5458-5459	18782	5416	9052	4762	36	34	56	43	2	24	1	1
libtiff-ee2ce5-b5691a	3311995	2609366	126968	113041	328	328	328	328	1	4478	1	1
php-310991-310999	802073	185001	45417	44536	1	1	1	1	1	19354	1	1
php-308734-308761	464785	151934	43563	38707	0	0	0	0	0	46137	-	-
php-308262-308315	658758	192659	58189	56960	1	1	2	2	1	8635	1	1
php-307562-307561	651312	163053	60650	53423	0	0	0	0	0	38971	-	-
php-309579-309580	629763	198376	64486	59261	2	2	2	2	1	8104	1	1
php-310011-310050	599295	162849	666	666	0	0	0	0	0	16975	-	-
php-309688-309716	621923	191952	47000	46831	31	31	32	32	0	46337	-	-
php-309516-309535	561904	136030	34972	27427	0	0	0	0	0	42385	-	-
php-307846-307853	541823	130366	32871	26033	0	0	0	0	0	45315	-	-
php-311346-311348	1404751	123671	5524	5524	38	38	43	43	2	411	1	1
php-307914-307915	549783	175191	55642	48600	1	0	1	0	1	3	1	1
php-309111-309159	539576	178631	58288	50205	1	1	1	1	0	39224	-	-
php-309892-309910	520768	177309	46811	45988	17	17	22	22	3	15340	1	1

Table A.15: Prophet-2000-RExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	780425	141182	39259	15947	27	24	54	51	0	87034	-	-
libtiff-5b0217-3dfb33	2027218	1352092	22673	22656	111	111	2178	2178	0	112099	-	-
libtiff-d13be7-ccadf4	2414317	1771135	37308	37266	93	93	3371	3258	2	4324	1	1
lighttpd-2661-2662	1290186	1016413	104869	90483	69	23	102	56	0	1217473	-	-
lighttpd-1913-1914	1206699	935121	57257	54013	42	42	109	109	0	86441	-	-
python-69934-69935	937134	386935	34092	28463	0	0	0	0	0	55996	-	-
gmp-13420-13421	823835	104714	60747	16845	4	3	19	18	1	48675	4	19
gzip-a1d3d4-f17cbd	399573	92972	127959	34611	14	0	14	0	1	6336	2	2
python-70056-70059	573639	236707	5048	5044	0	0	0	0	0	8292	-	-
fbc-5458-5459	18782	5416	6972	4007	36	34	120	89	0	24	-	-
libtiff-ee2ce5-b5691a	3311995	2609366	48940	48555	90	90	1952	1952	1	4478	1	1
php-310991-310999	802073	185001	45417	44536	1	1	1	1	1	19354	1	1
php-308734-308761	464785	151934	43596	38707	0	0	0	0	0	46137	-	-
php-308262-308315	658758	192659	14190	14158	9	9	21	21	1	8635	2	2
php-307562-307561	651312	163053	60650	53423	0	0	0	0	0	38971	-	-
php-309579-309580	629763	198376	35100	34932	11	11	38	38	1	8104	6	11
php-310011-310050	599295	162849	666	666	0	0	0	0	0	16975	-	-
php-309688-309716	621923	191952	7147	7120	13	13	40	40	0	46337	-	-
php-309516-309535	561904	136030	34838	27396	0	0	0	0	0	42385	-	-
php-307846-307853	541823	130366	32846	26033	0	0	0	0	0	45315	-	-
php-311346-311348	1404751	123671	5524	5524	3	3	42	42	1	411	1	1
php-307914-307915	549783	175191	55601	48600	1	0	1	0	1	3	1	1
php-309111-309159	539576	178631	58271	50205	1	1	1	1	0	39224	-	-
php-309892-309910	520768	177309	11497	11492	11	11	39	39	1	15340	1	1

Table A.16: Prophet-2000-RExt-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	10949	3394	10949	3394	33	28	111	106	0	-	-	-
libtiff-5b0217-3dfb33	83937	19298	64840	19298	174	147	177	150	0	-	-	-
libtiff-d13be7-ccadf4	184068	141307	45961	45374	1723	1723	2003	1723	1	2408	3	3
lighttpd-2661-2662	34688	27541	34688	27541	70	8	73	11	0	-	-	-
lighttpd-1913-1914	32903	21707	32903	21707	4	4	4	4	0	-	-	-
python-69934-69935	17828	5874	6960	4981	0	0	0	0	0	-	-	-
gmp-13420-13421	29703	5319	29703	5319	0	0	0	0	0	-	-	-
gzip-a1d3d4-f17cbd	25303	7535	25303	7535	5	0	5	0	1	12083	4	4
python-70056-70059	22537	9353	4254	3971	0	0	0	0	0	-	-	-
fbc-5458-5459	4864	2144	4864	2144	26	26	46	35	2	273	8	11
libtiff-ee2ce5-b5691a	34863	25186	34863	25186	328	328	328	328	1	3692	1	1
php-310991-310999	47688	11762	27894	11762	2	2	2	2	2	1348	1	1
php-308734-308761	14	11	14	11	0	0	0	0	0	-	-	-
php-308262-308315	22003	1869	5933	1869	0	0	0	0	0	2176	-	-
php-307562-307561	21252	4869	11682	4869	1	0	1	0	1	3224	1	1
php-309579-309580	41568	6890	22390	6890	2	2	2	2	1	46	1	1
php-310011-310050	34470	8642	4739	3465	52	6	54	9	0	13476	-	-
php-309688-309716	47600	11980	7073	5111	65	64	67	66	0	5240	-	-
php-309516-309535	19142	4578	19142	4578	1	0	1	0	1	2584	1	1
php-307846-307853	13971	3009	13971	3009	1	0	1	0	1	2570	1	1
php-311346-311348	8096	3027	8096	3027	50	38	72	60	2	465	1	1
php-307914-307915	25707	7731	25707	7731	1	0	1	0	1	3135	1	1
php-309111-309159	32541	6583	22565	6583	10	1	10	1	1	15538	10	10
php-309892-309910	8665	1432	8665	1432	21	17	26	22	4	92	1	1

Table A.17: SPR-100 Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	10949	3394	10949	3394	29	24	56	51	0	-	-	-
libtiff-5b0217-3dfb33	83937	19298	13487	5683	162	162	2879	2879	0	-	-	-
libtiff-d13be7-ccadf4	184068	141307	5269	5269	247	247	4450	4450	2	2408	5	104
lighttpd-2661-2662	34688	27541	34688	27541	78	16	109	47	0	-	-	-
lighttpd-1913-1914	32903	21707	32903	21707	4	4	12	12	0	-	-	-
python-69934-69935	17828	5874	6963	4984	0	0	0	0	0	-	-	-
gmp-13420-13421	29703	5319	29703	5319	0	0	0	0	0	-	-	-
gzip-a1d3d4-f17cbd	25303	7535	25303	7535	5	0	5	0	1	12083	4	4
python-70056-70059	22537	9353	4773	4353	0	0	0	0	0	-	-	-
fbc-5458-5459	4864	2144	4864	2144	23	23	98	71	0	273	-	-
libtiff-ee2ce5-b5691a	34863	25186	4828	4185	101	101	3829	3829	1	3692	1	1
php-310991-310999	47688	11762	27444	11762	2	2	2	2	2	1348	1	1
php-308734-308761	14	11	14	11	0	0	0	0	0	-	-	-
php-308262-308315	22003	1869	2416	784	12	12	34	34	1	2176	12	25
php-307562-307561	21252	4869	11831	4869	1	0	1	0	1	3224	1	1
php-309579-309580	41568	6890	15680	6890	11	11	38	38	1	46	1	1
php-310011-310050	34470	8642	4739	3465	52	6	58	13	0	13476	-	-
php-309688-309716	47600	11980	6235	4276	15	14	70	69	0	5240	-	-
php-309516-309535	19142	4578	19142	4578	1	0	1	0	1	2584	1	1
php-307846-307853	13971	3009	13971	3009	1	0	1	0	1	2570	1	1
php-311346-311348	8096	3027	1030	1022	31	30	102	101	2	465	1	1
php-307914-307915	25707	7731	25707	7731	1	0	1	0	1	3135	1	1
php-309111-309159	32541	6583	22886	6583	10	1	10	1	1	15538	10	10
php-309892-309910	8665	1432	353	187	11	11	97	97	1	92	1	1

Table A.18: SPR-100-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	in Plausible
gmp-14166-14167	12718	3394	12718	3394	33	28	111	106	0	-	-	-
libtiff-5b0217-3dfb33	88613	19298	68340	19298	174	147	177	150	0	-	-	-
libtiff-d13be7-ccadf4	186777	141307	160494	141307	1723	1723	2003	1723	1	2117	3	3
lighttpd-2661-2662	35810	27541	35810	27541	66	4	66	4	0	-	-	-
lighttpd-1913-1914	34917	21707	34917	21707	4	4	4	4	0	-	-	-
python-69934-69935	18168	5874	7329	5320	0	0	0	0	0	-	-	-
gmp-13420-13421	49743	5319	49743	5319	0	0	0	0	0	-	-	-
gzip-a1d3d4-f17cbd	31251	7535	31251	7535	5	0	5	0	1	17958	4	4
python-70056-70059	26398	9353	4794	4365	0	0	0	0	0	-	-	-
fbc-5458-5459	5938	2144	5938	2144	28	26	48	35	2	264	8	11
libtiff-ee2ce5-b5691a	37661	25186	37661	25186	328	328	328	328	1	3680	1	1
php-310991-310999	48843	11762	27311	11762	2	2	2	2	2	1289	1	1
php-308734-308761	14	11	14	11	0	0	0	0	0	-	-	-
php-308262-308315	22472	1869	3304	1211	2	2	4	4	1	2177	2	3
php-307562-307561	21944	4869	11937	4869	1	0	1	0	1	3227	1	1
php-309579-309580	42322	6890	22265	6890	2	2	2	2	1	46	1	1
php-310011-310050	35256	8642	4689	3465	52	6	55	9	0	14262	-	-
php-309688-309716	48118	11980	7083	5111	63	62	64	63	1	5240	34	35
php-309516-309535	21008	4578	21008	4578	1	0	1	0	1	2583	1	1
php-307846-307853	16435	3009	16435	3009	1	0	1	0	1	2616	1	1
php-311346-311348	8173	3027	8173	3027	51	38	73	60	2	455	1	1
php-307914-307915	27121	7731	27121	7731	1	0	1	0	1	3135	1	1
php-309111-309159	34080	6583	23959	6583	10	1	10	1	1	17075	8	8
php-309892-309910	10855	1432	2556	1432	69	17	74	22	3	224	1	1

Table A.19: SPR-100-RExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	12718	3394	12718	3394	29	24	56	51	0	-	-	-
libtiff-5b0217-3dfb33	88613	19298	10264	5607	136	136	2721	2721	0	-	-	-
libtiff-d13be7-ccadf4	186777	141307	5269	5269	232	232	4350	4350	2	2117	5	104
lighttpd-2661-2662	35810	27541	35810	27541	74	12	97	35	0	-	-	-
lighttpd-1913-1914	34917	21707	34917	21707	4	4	12	12	0	-	-	-
python-69934-69935	18168	5874	7047	5066	0	0	0	0	0	-	-	-
gmp-13420-13421	49743	5319	49743	5319	0	0	0	0	0	-	-	-
gzip-a1d3d4-f17cbd	31251	7535	31251	7535	5	0	5	0	1	17958	4	4
python-70056-70059	26398	9353	4825	4365	0	0	0	0	0	-	-	-
fbc-5458-5459	5938	2144	5938	2144	25	23	100	71	0	264	-	-
libtiff-ee2ce5-b5691a	37661	25186	4765	4121	101	101	3851	3851	1	3680	1	1
php-310991-310999	48843	11762	27441	11762	2	2	2	2	2	1289	1	1
php-308734-308761	14	11	14	11	0	0	0	0	0	-	-	-
php-308262-308315	22472	1869	2416	784	16	16	39	39	1	2177	12	25
php-307562-307561	21944	4869	11967	4869	1	0	1	0	1	3227	1	1
php-309579-309580	42322	6890	15360	6890	11	11	38	38	1	46	1	1
php-310011-310050	35256	8642	4679	3465	48	6	53	13	0	14262	-	-
php-309688-309716	48118	11980	6235	4276	15	14	68	67	0	5240	-	-
php-309516-309535	21008	4578	21008	4578	1	0	1	0	1	2583	1	1
php-307846-307853	16435	3009	16435	3009	1	0	1	0	1	2616	1	1
php-311346-311348	8173	3027	1040	1022	32	30	105	103	2	455	1	1
php-307914-307915	27121	7731	27121	7731	1	0	1	0	1	3135	1	1
php-309111-309159	34080	6583	23843	6583	10	1	10	1	1	17075	8	8
php-309892-309910	10855	1432	353	187	11	11	102	102	1	224	1	1

Table A.20: SPR-100-RExt-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	39287	14161	33154	14161	34	28	112	106	0	-	-	-
libtiff-5b0217-3dfb33	221134	65017	107534	65017	237	149	240	152	1	56644	208	211
libtiff-d13be7-ccadf4	296426	228659	230917	214697	1723	1723	2003	1723	1	372	3	3
lighttpd-2661-2662	120263	98028	105423	98028	52	4	55	7	0	-	-	-
lighttpd-1913-1914	68199	48133	45191	40674	55	55	55	55	0	-	-	-
python-69934-69935	47544	13775	11024	8634	0	0	0	0	0	-	-	-
gmp-13420-13421	50672	8763	50672	8763	3	0	3	0	2	14645	1	1
gzip-a1d3d4-f17cbd	48702	15104	48702	15104	14	0	14	0	1	21926	4	4
python-70056-70059	39599	17961	4126	3652	0	0	0	0	0	-	-	-
fbc-5458-5459	9857	4495	9791	4495	37	37	61	46	2	454	8	11
libtiff-ee2ce5-b5691a	171379	106867	14068	13454	15	15	15	15	1	13296	1	1
php-310991-310999	89230	18988	31084	16653	2	2	2	2	2	384	1	1
php-308734-308761	14692	4160	14692	4160	4	4	4	4	2	5771	1	1
php-308262-308315	90431	10845	9137	8516	0	0	0	0	0	7191	-	-
php-307562-307561	31597	6997	13425	6997	1	0	1	0	1	4918	1	1
php-309579-309580	60351	11416	25400	11416	2	2	2	2	1	46	1	1
php-310011-310050	77671	16558	8160	7316	32	32	49	49	0	30647	-	-
php-309688-309716	71633	15744	10699	6516	31	30	32	31	0	8398	-	-
php-309516-309535	27098	6314	27098	6314	1	0	1	0	1	4000	1	1
php-307846-307853	22131	4757	21654	4757	1	0	1	0	1	3867	1	1
php-311346-311348	9799	3879	9799	3879	50	38	72	60	2	312	1	1
php-307914-307915	47988	15066	43285	15066	1	0	1	0	1	5748	1	1
php-309111-309159	52908	12232	29459	12232	10	1	10	1	1	24347	10	10
php-309892-309910	40758	9999	17455	9999	17	17	22	22	3	179	1	1

Table A.21: SPR-200 Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	39287	14161	34474	14161	30	24	57	51	0	-	-	-
libtiff-5b0217-3dfb33	221134	65017	13644	8420	111	111	2604	2604	0	56644	-	-
libtiff-d13be7-ccadf4	296426	228659	13486	13486	225	225	4240	4240	2	372	5	104
lighttpd-2661-2662	120263	98028	105296	98028	36	12	57	33	0	-	-	-
lighttpd-1913-1914	68199	48133	41374	37396	49	49	125	125	0	-	-	-
python-69934-69935	47544	13775	11862	9464	0	0	0	0	0	-	-	-
gmp-13420-13421	50672	8763	50672	8763	3	0	3	0	2	14645	1	1
gzip-a1d3d4-f17cbd	48702	15104	48702	15104	14	0	14	0	1	21926	4	4
python-70056-70059	39599	17961	4364	3836	0	0	0	0	0	-	-	-
fbc-5458-5459	9857	4495	7000	4495	34	34	120	89	0	454	-	-
libtiff-ee2ce5-b5691a	171379	106867	16480	14448	101	101	3603	3603	1	13296	1	1
php-310991-310999	89230	18988	31084	16653	2	2	2	2	2	384	1	1
php-308734-308761	14692	4160	14692	4160	4	4	4	4	2	5771	1	1
php-308262-308315	90431	10845	9167	8516	0	0	0	0	0	7191	-	-
php-307562-307561	31597	6997	13565	6997	1	0	1	0	1	4918	1	1
php-309579-309580	60351	11416	17487	11416	11	11	38	38	1	46	1	1
php-310011-310050	77671	16558	10005	7386	21	20	38	37	0	30647	-	-
php-309688-309716	71633	15744	9861	5681	15	14	38	37	0	8398	-	-
php-309516-309535	27098	6314	27098	6314	1	0	1	0	1	4000	1	1
php-307846-307853	22131	4757	21674	4757	1	0	1	0	1	3867	1	1
php-311346-311348	9799	3879	1329	1319	31	30	104	103	2	312	1	1
php-307914-307915	47988	15066	43355	15066	1	0	1	0	1	5748	1	1
php-309111-309159	52908	12232	29464	12232	10	1	10	1	1	24347	10	10
php-309892-309910	40758	9999	4986	4931	11	11	92	92	1	179	1	1

Table A.22: SPR-200-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	61171	14161	50781	14161	33	28	111	106	0	-	-	-
libtiff-5b0217-3dfb33	278857	65017	104524	65017	1164	149	1167	152	1	56646	209	212
libtiff-d13be7-ccadf4	300745	228659	226189	210869	1723	1723	2003	1723	1	6080	3	3
lighttpd-2661-2662	124449	98028	108906	98028	9	3	10	4	0	-	-	-
lighttpd-1913-1914	70712	48133	45056	40544	55	55	55	55	0	-	-	-
python-69934-69935	48560	13775	11024	8634	0	0	0	0	0	24561	-	-
gmp-13420-13421	79763	8763	74674	8763	2	0	2	0	2	40506	1	1
gzip-a1d3d4-f17cbd	58432	15104	58432	15104	14	0	14	0	1	31657	4	4
python-70056-70059	43598	17961	4364	3836	0	0	0	0	0	-	-	-
fbc-5458-5459	11828	4495	10288	4495	39	37	63	46	2	573	8	11
libtiff-ee2ce5-b5691a	190629	106867	167664	106867	328	328	328	328	1	13296	1	1
php-310991-310999	90936	18988	30573	16385	2	2	2	2	2	403	1	1
php-308734-308761	18784	4160	18784	4160	4	4	4	4	2	5728	1	1
php-308262-308315	92459	10845	9197	8516	0	0	0	0	0	7366	-	-
php-307562-307561	32546	6997	13573	6997	1	0	1	0	1	4918	1	1
php-309579-309580	61407	11416	24205	11416	2	2	2	2	1	46	1	1
php-310011-310050	78981	16558	7960	7316	32	32	49	49	0	31956	-	-
php-309688-309716	73424	15744	10699	6516	32	31	33	32	0	8398	-	-
php-309516-309535	29514	6314	29514	6314	1	0	1	0	1	3999	1	1
php-307846-307853	25447	4757	24349	4757	1	0	1	0	1	3868	1	1
php-311346-311348	9978	3879	9568	3879	52	38	74	60	2	312	1	1
php-307914-307915	50442	15066	44514	15066	1	0	1	0	1	5748	1	1
php-309111-309159	58903	12232	32717	12232	10	1	10	1	1	30340	10	10
php-309892-309910	52975	9999	17434	9999	17	17	22	22	3	716	1	1

Table A.23: SPR-200-RExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	61171	14161	51587	14161	29	24	56	51	0	-	-	-
libtiff-5b0217-3dfb33	278857	65017	12964	8420	111	111	2604	2604	0	56646	-	-
libtiff-d13be7-ccadf4	300745	228659	13486	13486	232	232	4329	4329	2	6080	5	104
lighttpd-2661-2662	124449	98028	108266	98028	10	4	16	10	0	-	-	-
lighttpd-1913-1914	70712	48133	41374	37396	48	48	123	123	0	-	-	-
python-69934-69935	48560	13775	11024	8634	0	0	0	0	0	24561	-	-
gmp-13420-13421	79763	8763	74666	8763	2	0	2	0	2	40506	1	1
gzip-a1d3d4-f17cbd	58432	15104	58432	15104	14	0	14	0	1	31657	4	4
python-70056-70059	43598	17961	4364	3836	0	0	0	0	0	-	-	-
fbc-5458-5459	11828	4495	6799	4495	34	34	120	89	0	573	-	-
libtiff-ee2ce5-b5691a	190629	106867	16480	14448	101	101	3591	3591	1	13296	1	1
php-310991-310999	90936	18988	30423	16385	2	2	2	2	2	403	1	1
php-308734-308761	18784	4160	18784	4160	4	4	4	4	2	5728	1	1
php-308262-308315	92459	10845	9197	8516	0	0	0	0	0	7366	-	-
php-307562-307561	32546	6997	13518	6997	1	0	1	0	1	4918	1	1
php-309579-309580	61407	11416	14155	8539	11	11	36	36	1	46	1	1
php-310011-310050	78981	16558	9895	7316	21	20	38	37	0	31956	-	-
php-309688-309716	73424	15744	9861	5681	15	14	35	34	0	8398	-	-
php-309516-309535	29514	6314	29514	6314	1	0	1	0	1	3999	1	1
php-307846-307853	25447	4757	24329	4757	1	0	1	0	1	3868	1	1
php-311346-311348	9978	3879	1329	1319	31	30	105	104	2	312	1	1
php-307914-307915	50442	15066	44544	15066	1	0	1	0	1	5748	1	1
php-309111-309159	58903	12232	33002	12232	10	1	10	1	1	30340	10	10
php-309892-309910	52975	9999	4986	4931	11	11	89	89	1	716	1	1

Table A.24: SPR-200-RExt-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	59327	22055	41316	22055	34	28	112	106	0	-	-	-
libtiff-5b0217-3dfb33	276142	85251	123778	85251	237	149	240	152	1	59068	206	209
libtiff-d13be7-ccadf4	568172	432083	250826	237142	1723	1723	2003	1723	1	6126	3	3
lighttpd-2661-2662	191579	159862	119941	113050	10	5	13	8	0	-	-	-
lighttpd-1913-1914	159739	114667	37901	34739	32	32	32	32	0	-	-	-
python-69934-69935	65326	18300	11022	8243	0	0	0	0	0	-	-	-
gmp-13420-13421	69661	12794	58934	12794	15	5	15	5	2	20779	6	6
gzip-a1d3d4-f17cbd	82349	18024	82349	18024	14	0	14	0	1	30814	4	4
python-70056-70059	58530	26327	5006	4985	0	0	0	0	0	-	-	-
fbc-5458-5459	16353	5416	9850	5416	37	37	61	46	2	299	5	6
libtiff-ee2ce5-b5691a	218252	142593	203493	142593	328	328	328	328	1	20674	1	1
php-310991-310999	139311	31061	31955	19597	2	2	2	2	2	2256	1	1
php-308734-308761	30623	7437	20855	7437	4	4	4	4	2	7493	1	1
php-308262-308315	150963	19358	14708	14158	0	0	0	0	0	10111	-	-
php-307562-307561	60507	15748	18574	11458	1	0	1	0	1	10019	1	1
php-309579-309580	101940	17784	23578	15481	2	2	2	2	1	46	1	1
php-310011-310050	105542	23824	10543	10543	37	37	55	55	0	41309	-	-
php-309688-309716	95206	19747	13260	7432	7	6	8	7	0	10803	-	-
php-309516-309535	52377	12093	40324	12093	1	0	1	0	1	4695	1	1
php-307846-307853	40272	8051	20723	8051	3	2	4	3	1	5909	1	1
php-311346-311348	14543	5619	8568	5619	52	41	74	63	2	741	4	4
php-307914-307915	64378	20107	45432	20107	1	0	1	0	1	8122	1	1
php-309111-309159	86627	22335	41971	22335	2	1	2	1	0	39752	-	-
php-309892-309910	62347	19484	21735	15396	17	17	22	22	3	641	1	1

Table A.25: SPR-300 Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	59327	22055	42448	22055	30	24	57	51	0	-	-	-
libtiff-5b0217-3dfb33	276142	85251	16314	9388	111	111	2598	2598	0	59068	-	-
libtiff-d13be7-ccadf4	568172	432083	19425	19425	203	203	4089	4089	2	6126	5	104
lighttpd-2661-2662	191579	159862	116171	109330	14	9	34	29	0	-	-	-
lighttpd-1913-1914	159739	114667	28702	27455	24	24	53	53	0	107499	-	-
python-69934-69935	65326	18300	11077	8243	0	0	0	0	0	-	-	-
gmp-13420-13421	69661	12794	58745	12794	13	3	28	18	2	20779	4	19
gzip-a1d3d4-f17cbd	82349	18024	82349	18024	14	0	14	0	1	30814	4	4
python-70056-70059	58530	26327	5032	5011	0	0	0	0	0	835	-	-
fbc-5458-5459	16353	5416	6925	4707	34	34	120	89	0	299	-	-
libtiff-ee2ce5-b5691a	218252	142593	24635	22006	101	101	3244	3244	1	20674	1	1
php-310991-310999	139311	31061	32409	19972	2	2	2	2	2	2256	1	1
php-308734-308761	30623	7437	20902	7437	4	4	4	4	2	7493	1	1
php-308262-308315	150963	19358	14748	14158	0	0	0	0	0	10111	-	-
php-307562-307561	60507	15748	18594	11458	1	0	1	0	1	10019	1	1
php-309579-309580	101940	17784	12994	8849	11	11	38	38	1	46	1	1
php-310011-310050	105542	23824	10543	10543	23	23	54	54	0	41309	-	-
php-309688-309716	95206	19747	13260	7432	7	6	8	7	0	10803	-	-
php-309516-309535	52377	12093	41374	12093	1	0	1	0	1	4695	1	1
php-307846-307853	40272	8051	17882	8051	5	4	12	11	1	5909	1	1
php-311346-311348	14543	5619	1655	1644	34	33	97	96	2	741	4	4
php-307914-307915	64378	20107	45791	20107	1	0	1	0	1	8122	1	1
php-309111-309159	86627	22335	42158	22335	2	1	2	1	0	39752	-	-
php-309892-309910	62347	19484	4917	4917	11	11	85	85	1	641	1	1

Table A.26: SPR-300-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	95751	22055	54107	22055	39	28	117	106	1	50188	32	110
libtiff-5b0217-3dfb33	336808	85251	123207	85251	1164	149	1165	155	1	59066	209	215
libtiff-d13be7-ccadf4	573714	432083	87760	87073	309	309	309	309	1	4629	3	3
lighttpd-2661-2662	197676	159862	120127	113198	8	3	8	3	0	-	-	-
lighttpd-1913-1914	168946	114667	37891	34739	32	32	32	32	0	-	-	-
python-69934-69935	66616	18300	12534	9511	0	0	0	0	0	32709	-	-
gmp-13420-13421	107650	12794	78103	12794	41	5	41	5	1	46636	6	6
gzip-a1d3d4-f17cbd	100652	18024	91433	18024	14	0	14	0	1	49115	4	4
python-70056-70059	64192	26327	5032	5011	0	0	0	0	0	-	-	-
fbc-5458-5459	18782	5416	12524	5416	34	32	55	38	2	582	8	11
libtiff-ee2ce5-b5691a	241707	142593	195947	142593	328	328	328	328	1	20686	1	1
php-310991-310999	142164	31061	31855	19597	2	2	2	2	2	2139	1	1
php-308734-308761	36012	7437	21042	7437	4	4	4	4	2	7493	1	1
php-308262-308315	159696	19358	14748	14158	0	0	0	0	0	10106	-	-
php-307562-307561	64392	15748	18564	11458	1	0	1	0	1	10022	1	1
php-309579-309580	103151	17784	23738	15481	2	2	2	2	1	46	1	1
php-310011-310050	107479	23824	10543	10543	37	37	55	55	0	43288	-	-
php-309688-309716	97372	19747	13259	7432	10	9	11	10	0	10803	-	-
php-309516-309535	56549	12093	43485	12093	1	0	1	0	1	4694	1	1
php-307846-307853	44992	8051	21298	8051	3	2	4	3	1	5909	1	1
php-311346-311348	14889	5619	8607	5619	52	41	74	63	2	741	4	4
php-307914-307915	67286	20107	46603	20107	1	0	1	0	1	8122	1	1
php-309111-309159	102384	22335	43158	22335	2	1	2	1	0	55064	-	-
php-309892-309910	74780	19484	21370	15253	17	17	22	22	3	3305	1	1

Table A.27: SPR-300-RExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	95751	22055	54176	22055	35	24	62	51	1	50188	28	55
libtiff-5b0217-3dfb33	336808	85251	13784	9388	111	111	2604	2604	0	59066	-	-
libtiff-d13be7-ccadf4	573714	432083	19425	19425	215	215	4125	4125	2	4629	5	104
lighttpd-2661-2662	197676	159862	116171	109330	16	11	31	26	0	-	-	-
lighttpd-1913-1914	168946	114667	28116	26867	24	24	53	53	0	107498	-	-
python-69934-69935	66616	18300	12773	9592	0	0	0	0	0	32709	-	-
gmp-13420-13421	107650	12794	77443	12794	39	3	54	18	1	46636	4	19
gzip-a1d3d4-f17cbd	100652	18024	91870	18024	14	0	14	0	1	49115	4	4
python-70056-70059	64192	26327	5032	5011	0	0	0	0	0	3165	-	-
fbc-5458-5459	18782	5416	9253	5416	29	27	100	67	0	582	-	-
libtiff-ee2ce5-b5691a	241707	142593	24635	22006	101	101	3412	3412	1	20686	1	1
php-310991-310999	142164	31061	31955	19597	2	2	2	2	2	2139	1	1
php-308734-308761	36012	7437	21072	7437	4	4	4	4	2	7493	1	1
php-308262-308315	159696	19358	14748	14158	0	0	0	0	0	10106	-	-
php-307562-307561	64392	15748	18534	11458	1	0	1	0	1	10022	1	1
php-309579-309580	103151	17784	13274	8849	11	11	38	38	1	46	1	1
php-310011-310050	107479	23824	10543	10543	23	23	54	54	0	43288	-	-
php-309688-309716	97372	19747	12494	6667	7	7	13	13	0	10803	-	-
php-309516-309535	56549	12093	43728	12093	1	0	1	0	1	4694	1	1
php-307846-307853	44992	8051	20127	8051	5	4	12	11	1	5909	1	1
php-311346-311348	14889	5619	1655	1644	34	33	98	97	2	741	4	4
php-307914-307915	67286	20107	46743	20107	1	0	1	0	1	8122	1	1
php-309111-309159	102384	22335	41638	22335	2	1	2	1	0	55064	-	-
php-309892-309910	74780	19484	4917	4917	11	11	87	87	1	3305	1	1

Table A.28: SPR-300-RExt-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	448318	141182	48639	40229	28	28	106	106	0	-	-	-
libtiff-5b0217-3dfb33	1941177	1352092	376088	346011	59	59	59	59	0	309678	-	-
libtiff-d13be7-ccadf4	2353240	1771135	537943	510074	39	39	39	39	1	34935	3	3
lighttpd-2661-2662	1269307	1016413	158849	158849	84	84	108	108	1	73333	10	18
lighttpd-1913-1914	1183286	935121	67947	67947	30	30	30	30	0	-	-	-
python-69934-69935	863452	386935	74931	67619	0	0	0	0	0	-	-	-
gmp-13420-13421	389124	104714	77939	51103	5	5	5	5	0	148326	-	-
gzip-a1d3d4-f17cbd	341681	92972	172669	92972	15	0	15	0	1	21929	1	1
python-70056-70059	526778	236707	5549	5549	0	0	0	0	0	-	-	-
fbc-5458-5459	16353	5416	10804	5416	37	37	61	46	2	299	5	6
libtiff-ee2ce5-b5691a	3237374	2609366	559844	545779	328	328	328	328	1	510769	1	1
php-310991-310999	783770	185001	71978	71969	1	1	1	1	1	20811	1	1
php-308734-308761	440926	151934	70790	69209	0	0	0	0	0	70482	-	-
php-308262-308315	637558	192659	56294	56291	0	0	0	0	0	84098	-	-
php-307562-307561	637588	163053	85244	84684	0	0	0	0	0	85214	-	-
php-309579-309580	621880	198376	103822	99184	2	2	2	2	1	28250	1	1
php-310011-310050	582198	162849	4293	4293	0	0	0	0	0	249855	-	-
php-309688-309716	609953	191952	100081	94908	1	0	1	0	0	84053	-	-
php-309516-309535	496992	136030	56498	47458	0	0	0	0	0	58080	-	-
php-307846-307853	477171	130366	58099	49977	0	0	0	0	0	59502	-	-
php-311346-311348	1030763	123671	11552	11552	29	29	47	47	2	8838	1	1
php-307914-307915	539078	175191	89793	83690	1	0	1	0	1	57702	1	1
php-309111-309159	513188	178631	90328	84540	1	1	1	1	0	249184	-	-
php-309892-309910	498669	177309	90902	86247	5	5	5	5	1	4347	1	1

Table A.29: SPR-2000 Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	448318	141182	29653	24692	24	24	51	51	0	-	-	-
libtiff-5b0217-3dfb33	1941177	1352092	13873	13873	111	111	2442	2442	0	309678	-	-
libtiff-d13be7-ccadf4	2353240	1771135	288244	284634	52	52	2708	2708	2	34935	5	104
lighttpd-2661-2662	1269307	1016413	141022	141022	117	117	189	189	1	73333	9	14
lighttpd-1913-1914	1183286	935121	60084	60084	37	37	84	84	0	297232	-	-
python-69934-69935	863452	386935	75020	67619	0	0	0	0	0	-	-	-
gmp-13420-13421	389124	104714	69769	46070	3	3	18	18	0	148326	-	-
gzip-a1d3d4-f17cbd	341681	92972	173757	92972	15	0	15	0	1	21929	1	1
python-70056-70059	526778	236707	6816	6816	0	0	0	0	0	11955	-	-
fbc-5458-5459	16353	5416	6746	4640	34	34	120	89	0	299	-	-
libtiff-ee2ce5-b5691a	3237374	2609366	525373	513075	71	71	496	496	1	510769	1	1
php-310991-310999	783770	185001	71978	71969	1	1	1	1	1	20811	1	1
php-308734-308761	440926	151934	70790	69209	0	0	0	0	0	70482	-	-
php-308262-308315	637558	192659	56286	56283	0	0	0	0	0	84098	-	-
php-307562-307561	637588	163053	92960	84711	0	0	0	0	0	85214	-	-
php-309579-309580	621880	198376	81472	81463	11	11	34	34	1	28250	1	1
php-310011-310050	582198	162849	4293	4293	0	0	0	0	0	249855	-	-
php-309688-309716	609953	191952	99991	94908	1	0	1	0	0	84053	-	-
php-309516-309535	496992	136030	56588	47458	0	0	0	0	0	58080	-	-
php-307846-307853	477171	130366	58149	49977	0	0	0	0	0	59502	-	-
php-311346-311348	1030763	123671	11552	11552	28	28	46	46	2	8838	1	1
php-307914-307915	539078	175191	89763	83690	1	0	1	0	1	57702	1	1
php-309111-309159	513188	178631	90378	84540	1	1	1	1	0	249184	-	-
php-309892-309910	498669	177309	3691	3691	11	11	42	42	1	4347	1	1

Table A.30: SPR-2000-CExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	780425	141182	27515	23574	10	10	29	29	0	214233	-	-
libtiff-5b0217-3dfb33	2027218	1352092	353982	324268	59	59	59	59	0	309679	-	-
libtiff-d13be7-ccadf4	2414317	1771135	532541	506165	39	39	39	39	1	33699	3	3
lighttpd-2661-2662	1290186	1016413	157305	157305	84	84	105	105	1	73333	10	18
lighttpd-1913-1914	1206699	935121	67947	67947	31	31	31	31	0	-	-	-
python-69934-69935	937134	386935	75716	67619	0	0	0	0	0	470550	-	-
gmp-13420-13421	823835	104714	69833	46134	5	5	5	5	0	174177	-	-
gzip-a1d3d4-f17cbd	399573	92972	183189	92972	5	0	5	0	1	21936	1	1
python-70056-70059	573639	236707	5761	5761	0	0	0	0	0	-	-	-
fbc-5458-5459	18782	5416	14112	5416	34	32	55	38	2	582	8	11
libtiff-ee2ce5-b5691a	3311995	2609366	544755	531755	329	329	329	329	1	510781	1	1
php-310991-310999	802073	185001	71978	71969	1	1	1	1	1	13860	1	1
php-308734-308761	464785	151934	70780	69209	0	0	0	0	0	70482	-	-
php-308262-308315	658758	192659	54186	54183	0	0	0	0	0	84097	-	-
php-307562-307561	651312	163053	92759	84711	0	0	0	0	0	85216	-	-
php-309579-309580	629763	198376	103872	99184	2	2	2	2	1	28294	1	1
php-310011-310050	599295	162849	5151	5151	1	1	2	2	0	266951	-	-
php-309688-309716	621923	191952	99781	94908	1	0	1	0	0	83933	-	-
php-309516-309535	561904	136030	56528	47458	0	0	0	0	0	58080	-	-
php-307846-307853	541823	130366	58119	49977	0	0	0	0	0	59503	-	-
php-311346-311348	1404751	123671	6977	6977	28	28	46	46	2	3218	1	1
php-307914-307915	549783	175191	89743	83690	1	0	1	0	1	57702	1	1
php-309111-309159	539576	178631	90328	84540	1	1	1	1	0	275198	-	-
php-309892-309910	520768	177309	90793	86247	5	5	5	5	1	5439	1	1

Table A.31: SPR-2000-RExt Statistics

Defect	Search Space Templates		Evaluated Templates		Plausible Templates		Plausible Patches		Correct Patches	Correct Template Rank		Correct Patch Rank in Plausible
	All	Cond.	All	Cond.	All	Cond.	All	Cond.		In Space	In Plausible	
gmp-14166-14167	780425	141182	27505	23574	6	6	13	13	0	214233	-	-
libtiff-5b0217-3dfb33	2027218	1352092	37909	37909	111	111	2264	2264	0	309679	-	-
libtiff-d13be7-ccadf4	2414317	1771135	285546	283944	52	52	2708	2708	2	33699	5	104
lighttpd-2661-2662	1290186	1016413	136816	136816	117	117	189	189	1	73333	9	14
lighttpd-1913-1914	1206699	935121	64918	64918	28	28	65	65	0	297229	-	-
python-69934-69935	937134	386935	75590	67619	0	0	0	0	0	470550	-	-
gmp-13420-13421	823835	104714	69833	46134	3	3	18	18	0	174177	-	-
gzip-a1d3d4-f17cbd	399573	92972	183149	92972	5	0	5	0	1	21936	1	1
python-70056-70059	573639	236707	5825	5825	0	0	0	0	0	6175	-	-
fbc-5458-5459	18782	5416	8812	5416	27	27	98	67	0	582	-	-
libtiff-ee2ce5-b5691a	3311995	2609366	525373	513075	71	71	471	471	1	510781	1	1
php-310991-310999	802073	185001	71688	71679	1	1	1	1	1	13860	1	1
php-308734-308761	464785	151934	70770	69209	0	0	0	0	0	70482	-	-
php-308262-308315	658758	192659	54186	54183	0	0	0	0	0	84097	-	-
php-307562-307561	651312	163053	85204	84684	0	0	0	0	0	85216	-	-
php-309579-309580	629763	198376	54142	54139	11	11	37	37	1	28294	1	1
php-310011-310050	599295	162849	5151	5151	0	0	0	0	0	266951	-	-
php-309688-309716	621923	191952	99781	94908	1	0	1	0	0	83933	-	-
php-309516-309535	561904	136030	56548	47458	0	0	0	0	0	58080	-	-
php-307846-307853	541823	130366	58119	49977	0	0	0	0	0	59503	-	-
php-311346-311348	1404751	123671	6977	6977	28	28	45	45	2	3218	1	1
php-307914-307915	549783	175191	89723	83690	1	0	1	0	1	57702	1	1
php-309111-309159	539576	178631	90328	84540	1	1	1	1	0	275198	-	-
php-309892-309910	520768	177309	8224	8224	11	11	41	41	1	5439	1	1

Table A.32: SPR-2000-RExt-CExt Statistics

Appendix B

Genesis Per Defect Experimental Results

Tables B.1, B.2, and B.3 show the results of Genesis when we run Genesis using the combined search space containing patches for NP, OOB, and CC defects. Tables B.4, B.5, and B.6 present the results of Genesis using per-defect-type search spaces for all types of defects. Tables B.7, B.8, and B.9 present the results of Genesis using the combined search space with the condition synthesis technique for all types of defects. Tables B.10, B.11, and B.12 present the results of Genesis using the combined search space without the patch prioritization learning for all types of defects. Finally, Tables B.13, B.14, B.15, contain the results of running our formulation of the PAR templates on our benchmark sets.

Each table contains one line for each error of its defect type. The "Init. Time" column presents the amount of time required to initialize the search for that error. The "Search Space Size" column presents the size of the search space for that error, the "Explored Space Size" column presents the size of the search space that the algorithm explores within the five hour timeout, the "Search Time" column presents the amount of time spent exploring the space, and "Validated Patches" presents the number of candidate patches that validate (produce correct outputs for all test cases). The last three columns present statistics for the first generated correct patch, specifically how long it takes to generate the patch ("Generation Time"), the rank of the first correct patch in the sequence of validated patches ("Validated Rank"), and the rank of the correct patch in the sequence of candidate patches ("Space Rank"). For Tables B.7, B.8, and B.9, there is an extra column ("Cond. Synthesis Saved"), which presents the ratio of the explored candidate patches that are pruned away by the condition synthesis technique.

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	<1m	42832	42832	71m	32	6m	13	1126
caelum-stella	2d2dd9c	<1m	17440	17440	54m	26	<1m	1	1
caelum-stella	e73113f	<1m	17535	17535	55m	26	<1m	1	1
HikariCP	ce4ff92	3m	121103	45355	>5h	46	-	-	-
nutz	80e85d0	2m	378640	57586	>5h	0	-	-	-
spring-data-rest	aa28aeb	6m	79798	5911	>5h	20	39m	3	778
checkstyle	8381754	2m	380041	77289	>5h	25	10m	1	38
checkstyle	536bc20	2m	471302	82244	>5h	50	10m	1	10
checkstyle	aaf606e	2m	448997	81239	>5h	0	-	-	-
checkstyle	aa829d4	<1m	532656	86660	>5h	3	20m	2	598
jongo	f46f658	<1m	263930	23454	>5h	1	-	-	-
DataflowJavaSDK	c06125d	3m	55490	36408	>5h	7	3m	1	1
webmagic	ff2f588	<1m	109723	70229	>5h	1	-	-	-
javapoet	70b38e5	<1m	172908	55070	>5h	0	-	-	-
closure-compiler	9828574	3m	>1000000	54953	>5h	41	16m	1	5
truth	99b314e	<1m	79357	79357	88m	0	-	-	-
error-prone	3709338	2m	504557	34148	>5h	5	23m	2	96
javaslang	faf9ac2	<1m	>1000000	228578	232m	46	37m	2	76
Activiti	3d624a5	2m	255897	607	>5h	75	9m	4	20
spring-hateoas	48749e7	<1m	23075	23075	46m	54	<1m	1	1

Table B.1: Results of the Genesis Combined Search Space on NP Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
Bukkit	a91c4c6	<1m	294393	225756	>5h	7	11m	1	1
RoaringBitmap	29c6d59	4m	689635	34635	>5h	3	-	-	-
commons-lang	52b46e7	1m	128007	67062	>5h	0	-	-	-
HdrHistogram	db18018	<1m	482000	124967	>5h	203	-	-	-
spring-hateoas	29b4334	<1m	29952	29952	34m	0	-	-	-
wicket	b708e2b	6m	133116	41525	>5h	38	20m	1	1442
coveralls-maven-plugin	20490f6	<1m	5088	5088	12m	0	-	-	-
named-regexp	82bdfeb	<1m	0	0	<1m	0	-	-	-
jgit	929862f	2m	96541	96541	225m	3	36m	1	10190
jPOS	df400ac	2m	128162	128162	218m	18	16m	1	5335
httpcore	dd00a9e	2m	206919	9420	>5h	74	43m	12	1586
vectorz	2291d0d	<1m	177962	177962	261m	25	14m	1	1162
maven-shared	77937e1	2m	0	0	<1m	0	-	-	-

Table B.2: Results of the Genesis Combined Search Space on OOB Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
jade4j	dd47397	<1m	171777	61649	>5h	1	181m	1	34265
jade4j	114e886	<1m	340472	52402	>5h	2	212m	1	35614
HdrHistogram	030aac1	<1m	40726	40726	80m	230	7m	16	2452
pdfbox	93c0b69	<1m	148930	120597	>5h	0	-	-	-
tree-root	fef0f36	<1m	28688	28688	21m	0	-	-	-
spoon	48d3126	8m	0	0	<1m	0	-	-	-
pebble	942aa6e	2m	151427	27058	>5h	0	-	-	-
fastjson	c886874	1m	>1000000	176898	>5h	0	-	-	-
htmlelements	bf3f275	1m	157745	107844	>5h	32	34m	1	7437
spring-cloud-connectors	56c6eca	<1m	101598	101598	182m	4	-	-	-
joinmo	a5ee885	<1m	454141	137112	>5h	16	-	-	-
buildergenerator	d9d73b3	<1m	44946	44946	84m	0	-	-	-
mybatis-3	809c35d	6m	793166	70895	>5h	0	-	-	-
antlr4	9e7b131	3m	5177	5177	72m	1	-	-	-
hamcrest-bean	84586d9	<1m	973459	117037	>5h	9	16m	1	2586
raml-java-parser	49aab8f	<1m	58871	58871	181m	0	-	-	-

Table B.3: Results of the Genesis Combined Search Space on CC Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	<1m	30599	30599	60m	13	2m	3	371
caelum-stella	2d2dd9c	<1m	11901	11901	37m	12	<1m	1	1
caelum-stella	e73113f	<1m	11964	11964	37m	12	<1m	1	1
HikariCP	ce4ff92	3m	188369	49250	>5h	30	-	-	-
nutz	80e85d0	1m	407660	123079	>5h	0	-	-	-
spring-data-rest	aa28aeb	7m	36286	5882	>5h	18	121m	3	2502
checkstyle	8381754	2m	641676	161122	>5h	6	14m	1	31
checkstyle	536bc20	2m	805920	168444	>5h	47	13m	1	2
checkstyle	aaf606e	2m	673428	162822	>5h	0	-	-	-
checkstyle	aa829d4	1m	732293	207175	>5h	7	17m	1	4877
jongo	f46f658	<1m	153232	41193	>5h	2	-	-	-
DataflowJavaSDK	c06125d	3m	33251	30586	>5h	30	3m	1	2
webmagic	ff2f588	<1m	756186	756186	216m	0	-	-	-
javapoet	70b38e5	<1m	239560	140194	>5h	0	-	-	-
closure-compiler	9828574	3m	227516	38366	>5h	6	18m	1	1
truth	99b314e	<1m	27516	27516	42m	0	-	-	-
error-prone	3709338	2m	400008	43924	>5h	2	27m	1	1
javaslang	faf9ac2	<1m	>1000000	323786	>5h	150	28m	1	6
Activiti	3d624a5	2m	649756	279	>5h	80	7m	1	3
spring-hateoas	48749e7	<1m	11073	11073	31m	25	<1m	1	1

Table B.4: Results of the Genesis Per-Defect-Type Search Space on NP Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
Bukkit	a91c4c6	<1m	129491	129491	140m	4	4m	1	4
RoaringBitmap	29c6d59	4m	223889	15809	>5h	0	-	-	-
commons-lang	52b46e7	1m	28310	28310	95m	0	-	-	-
HdrHistogram	db18018	<1m	83521	83521	222m	681	-	-	-
spring-hateoas	29b4334	<1m	9732	9732	9m	0	-	-	-
wicket	b708e2b	5m	28340	28340	153m	22	23m	2	2978
coveralls-maven-plugin	20490f6	<1m	764	764	1m	0	-	-	-
named-regexp	82bdfeb	<1m	0	0	<1m	0	-	-	-
jgit	929862f	2m	42419	42419	146m	6	38m	1	9300
jPOS	df400ac	2m	37179	37179	65m	9	3m	1	283
httpcore	dd00a9e	2m	70054	22075	>5h	134	120m	82	3160
vectorz	2291d0d	<1m	67356	67356	78m	33	14m	14	7622
maven-shared	77937e1	2m	0	0	<1m	0	-	-	-

Table B.5: Results of the Genesis Per-Defect-Type Search Space on OOB Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
jade4j	dd47397	<1m	75303	65173	>5h	1	125m	1	25608
jade4j	114e886	<1m	118087	69932	>5h	1	70m	1	13090
HdrHistogram	030aac1	<1m	19026	19026	38m	126	3m	11	287
pdfbox	93c0b69	<1m	150408	150408	269m	0	-	-	-
tree-root	fef0f36	<1m	22941	22941	4m	0	-	-	-
spoon	48d3126	8m	0	0	<1m	0	-	-	-
pebble	942aa6e	2m	190363	78037	>5h	0	-	-	-
fastjson	c886874	1m	664669	192976	>5h	0	-	-	-
htmlelements	bf3f275	1m	93142	93142	201m	0	-	-	-
spring-cloud-connectors	56c6eca	<1m	100925	100925	162m	0	-	-	-
joinmo	a5ee885	<1m	461121	101044	>5h	0	-	-	-
buildergenerator	d9d73b3	<1m	30074	30074	52m	0	-	-	-
mybatis-3	809c35d	6m	541027	61870	>5h	0	-	-	-
antlr4	9e7b131	3m	1329	1329	16m	0	-	-	-
hamcrest-bean	84586d9	<1m	548771	101882	>5h	3	-	-	-
raml-java-parser	49aab8f	<1m	26394	26394	125m	0	-	-	-

Table B.6: Results of the Genesis Per-Defect-Type Search Space on CC Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Cond. Synthesis Saved	Search Time	Validated Patches	First Correct Patch		
								Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	<1m	42832	42832	1.8%	218m	32	12m	13	1126
caelum-stella	2d2dd9c	<1m	17440	17440	13.4%	98m	26	<1m	1	1
caelum-stella	e73113f	<1m	17535	17535	13.9%	98m	26	<1m	1	1
HikariCP	ce4ff92	3m	121103	32758	15.2%	>5h	28	-	-	-
nutz	80e85d0	14m	378640	29975	6.9%	>5h	0	-	-	-
spring-data-rest	aa28aeb	6m	79798	4904	4.7%	>5h	19	45m	3	778
checkstyle	8381754	2m	380040	30510	6.8%	>5h	25	9m	1	38
checkstyle	536bc20	2m	471302	30167	4.4%	>5h	50	10m	1	10
checkstyle	aaf606e	2m	448997	35430	3.5%	>5h	0	-	-	-
checkstyle	aa829d4	1m	532656	45650	2.4%	>5h	3	13m	1	514
jongo	f46f658	<1m	263926	12685	4.2%	>5h	0	-	-	-
DataflowJavaSDK	c06125d	3m	55490	25008	7.5%	>5h	7	3m	1	1
webmagic	ff2f588	<1m	109723	36193	5.0%	>5h	0	-	-	-
javapoet	70b38e5	<1m	172908	33535	1.2%	>5h	0	-	-	-
closure-compiler	9828574	3m	>1000000	18698	12.4%	>5h	29	16m	1	5
truth	99b314e	<1m	79357	79357	10.3%	237m	0	-	-	-
error-prone	3709338	2m	504557	23547	15.4%	>5h	5	26m	2	96
javaslang	faf9ac2	<1m	>1000000	64195	16.4%	>5h	46	39m	2	76
Activiti	3d624a5	2m	255897	329	5.2%	>5h	28	9m	4	20
spring-hateoas	48749e7	<1m	23075	8369	17.6%	>5h	50	<1m	1	1

Table B.7: Results of the Genesis Combined Search Space with Condition Synthesis on NP Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Cond. Synthesis Saved	Search Time	Validated Patches	First Correct Patch		
								Generation Time	Validated Rank	Space Rank
Bukkit	a91c4c6	<1m	294393	113659	7.2%	>5h	7	12m	1	1
RoaringBitmap	29c6d59	4m	689635	24924	16.6%	>5h	2	-	-	-
commons-lang	52b46e7	1m	127997	30057	9.1%	>5h	0	-	-	-
HdrHistogram	db18018	<1m	482000	42542	16.8%	>5h	85	-	-	-
spring-hateoas	29b4334	<1m	29955	29955	15.5%	163m	0	-	-	-
wicket	b708e2b	6m	133116	27302	10.1%	>5h	31	26m	1	1442
coveralls-maven-plugin	20490f6	<1m	5088	5088	10.4%	41m	0	-	-	-
named-regexp	82bdfeb	<1m	0	0	0.0%	<1m	0	-	-	-
jgit	929862f	2m	96541	57597	7.2%	>5h	3	67m	1	10190
jPOS	df400ac	2m	128162	64165	6.8%	>5h	17	33m	1	5335
httpcore	dd00a9e	2m	206919	8486	11.1%	>5h	61	51m	11	1586
vectorz	2291d0d	<1m	177962	73628	4.6%	>5h	25	22m	1	1162
maven-shared	77937e1	2m	0	0	0.0%	<1m	0	-	-	-

Table B.8: Results of the Genesis Combined Search Space with Condition Synthesis on OOB Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Cond. Synthesis Saved	Search Time	Validated Patches	First Correct Patch		
								Generation Time	Validated Rank	Space Rank
jade4j	dd47397	<1m	171777	32430	11.7%	>5h	0	-	-	-
jade4j	114e886	<1m	340472	23294	4.0%	>5h	0	-	-	-
HdrHistogram	030aac1	<1m	40726	40726	2.0%	212m	224	16m	16	2452
pdfbox	93c0b69	<1m	148930	55369	4.3%	>5h	0	-	-	-
tree-root	fef0f36	<1m	28688	28688	3.6%	51m	0	-	-	-
spoon	48d3126	8m	0	0	0.0%	<1m	0	-	-	-
pebble	942aa6e	2m	151427	23380	13.0%	>5h	0	-	-	-
fastjson	c886874	1m	>1000000	100927	9.3%	>5h	0	-	-	-
htmlelements	bf3f275	1m	157745	33997	2.9%	>5h	32	61m	1	7437
spring-cloud-connectors	56c6eca	<1m	101598	59891	1.6%	>5h	2	-	-	-
joinmo	a5ee885	<1m	454159	68622	7.8%	>5h	16	-	-	-
buildergenerator	d9d73b3	<1m	44946	44946	1.0%	249m	0	-	-	-
mybatis-3	809c35d	7m	793166	37169	12.1%	>5h	0	-	-	-
antlr4	9e7b131	3m	5177	5177	2.4%	91m	1	-	-	-
hamcrest-bean	84586d9	<1m	973459	32067	7.7%	>5h	9	48m	1	2586
raml-java-parser	49aab8f	<1m	58871	42283	6.6%	>5h	0	-	-	-

Table B.9: Results of the Genesis Combined Search Space with Condition Synthesis on CC Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	<1m	42832	42832	73m	32	1m	1	14
caelum-stella	2d2dd9c	<1m	17440	17440	54m	26	<1m	1	43
caelum-stella	e73113f	<1m	17535	17535	55m	26	<1m	1	42
HikariCP	ce4ff92	3m	121103	45560	>5h	63	-	-	-
nutz	80e85d0	2m	378640	59464	>5h	0	-	-	-
spring-data-rest	aa28aeb	6m	79798	7375	>5h	13	217m	1	4917
checkstyle	8381754	2m	380041	65304	>5h	36	46m	5	3415
checkstyle	536bc20	2m	471302	71087	>5h	66	26m	1	13
checkstyle	aaf606e	2m	448997	72704	>5h	0	-	-	-
checkstyle	aa829d4	1m	532656	78152	>5h	0	-	-	-
jongo	f46f658	<1m	263926	33203	>5h	2	-	-	-
DataflowJavaSDK	c06125d	3m	55490	36849	>5h	7	15m	1	2202
webmagic	ff2f588	<1m	109723	71998	>5h	1	-	-	-
javapoet	70b38e5	<1m	172908	53849	>5h	0	-	-	-
closure-compiler	9828574	3m	>1000000	56226	>5h	41	16m	1	10
truth	99b314e	<1m	79357	79357	88m	0	-	-	-
error-prone	3709338	2m	504557	33673	>5h	5	57m	1	168
javaslang	faf9ac2	<1m	>1000000	888480	235m	46	38m	8	2209
Activiti	3d624a5	2m	255897	283	>5h	7	10m	2	250
spring-hateoas	48749e7	<1m	23075	23075	47m	54	<1m	1	21

Table B.10: Results of the Genesis Combined Search Space without Patch Prioritization Learning on NP Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
Bukkit	a91c4c6	<1m	294393	249060	>5h	7	11m	1	378
RoaringBitmap	29c6d59	4m	689635	41166	>5h	5	-	-	-
commons-lang	52b46e7	1m	128007	68634	>5h	0	-	-	-
HdrHistogram	db18018	<1m	481997	115144	>5h	221	-	-	-
spring-hateoas	29b4334	<1m	29952	29952	35m	0	-	-	-
wicket	b708e2b	6m	133116	39445	>5h	36	178m	12	26014
coveralls-maven-plugin	20490f6	<1m	5088	5088	12m	0	-	-	-
named-regexp	82bdfeb	<1m	0	0	<1m	0	-	-	-
jgit	929862f	2m	96541	96541	226m	3	71m	1	27865
jPOS	df400ac	2m	128162	128162	223m	17	12m	1	3442
httpcore	dd00a9e	2m	206919	11171	>5h	169	9m	1	101
vectorz	2291d0d	<1m	177960	177960	265m	25	11m	1	2
maven-shared	77937e1	2m	0	0	<1m	0	-	-	-

Table B.11: Results of the Genesis Combined Search Space without Patch Prioritization Learning on OOB Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
jade4j	dd47397	<1m	171777	62053	>5h	1	217m	1	43466
jade4j	114e886	<1m	340472	56015	>5h	0	-	-	-
HdrHistogram	030aac1	<1m	40726	40726	81m	230	7m	33	1588
pdfbox	93c0b69	<1m	148931	117382	>5h	0	-	-	-
tree-root	fef0f36	<1m	28688	28688	21m	0	-	-	-
spoon	48d3126	8m	0	0	<1m	0	-	-	-
pebble	942aa6e	2m	151427	35357	>5h	0	-	-	-
fastjson	c886874	1m	>1000000	240729	>5h	0	-	-	-
htmlelements	bf3f275	1m	157745	124437	>5h	32	7m	1	353
spring-cloud-connectors	56c6eca	<1m	101598	101598	182m	4	-	-	-
joinmo	a5ee885	<1m	454141	142234	>5h	16	-	-	-
buildergenerator	d9d73b3	<1m	44946	44946	84m	0	-	-	-
mybatis-3	809c35d	7m	793166	70237	>5h	0	-	-	-
antlr4	9e7b131	3m	5177	5177	72m	1	-	-	-
hamcrest-bean	84586d9	<1m	973459	125740	>5h	9	36m	1	6243
raml-java-parser	49aab8f	<1m	58871	58871	184m	0	-	-	-

Table B.12: Results of the Genesis Combined Search Space without Patch Prioritization Learning on CC Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
caelum-stella	2ec5459	1m	876	876	2m	4	<1m	1	5
caelum-stella	2d2dd9c	<1m	606	606	3m	9	<1m	1	18
caelum-stella	e73113f	<1m	614	614	3m	9	<1m	1	16
HikariCP	ce4ff92	3m	2021	2021	12m	1	-	-	-
nutz	80e85d0	2m	11937	11937	48m	9	-	-	-
spring-data-rest	aa28aeb	7m	1349	1349	59m	0	-	-	-
checkstyle	8381754	3m	8126	8126	33m	5	<1m	1	41
checkstyle	536bc20	2m	8551	8551	38m	10	<1m	1	6
checkstyle	aaf606e	2m	7862	7862	29m	0	-	-	-
checkstyle	aa829d4	<1m	8717	8717	33m	2	3m	1	606
jongo	f46f658	<1m	6395	6395	86m	3	-	-	-
DataflowJavaSDK	c06125d	3m	1519	1519	7m	0	-	-	-
webmagic	ff2f588	1m	4624	4624	15m	0	-	-	-
javapoet	70b38e5	<1m	3343	3343	17m	0	-	-	-
closure-compiler	9828574	3m	3809	3809	25m	2	1m	1	8
truth	99b314e	<1m	1128	1128	1m	0	-	-	-
error-prone	3709338	2m	15905	117	>5h	0	-	-	-
javaslang	faf9ac2	<1m	45225	45225	47m	0	-	-	-
Activiti	3d624a5	4m	6113	6113	286m	92	-	-	-
spring-hateoas	48749e7	<1m	357	357	1m	6	<1m	1	6

Table B.13: Results of PAR Templates on NP Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
Bukkit	a91c4c6	<1m	543	543	1m	2	<1m	1	9
RoaringBitmap	29c6d59	4m	2054	2054	24m	0	-	-	-
commons-lang	52b46e7	2m	1460	1460	3m	0	-	-	-
HdrHistogram	db18018	<1m	853	853	2m	0	-	-	-
spring-hateoas	29b4334	<1m	640	640	<1m	0	-	-	-
wicket	b708e2b	8m	2917	2917	18m	3	3m	1	550
coveralls-maven-plugin	20490f6	<1m	266	266	<1m	0	-	-	-
named-regexp	82bdfeb	<1m	0	0	<1m	0	-	-	-
jgit	929862f	3m	1234	1234	4m	0	-	-	-
jPOS	df400ac	3m	3171	3171	6m	0	-	-	-
httpcore	dd00a9e	2m	1588	1588	9m	4	2m	1	15
vectorz	2291d0d	<1m	2314	2314	5m	2	<1m	1	107
maven-shared	77937e1	2m	0	0	<1m	0	-	-	-

Table B.14: Results of PAR Templates on OOB Defects

Repository	Revision	Init. Time	Search Space Size	Explored Space Size	Search Time	Validated Patches	First Correct Patch		
							Generation Time	Validated Rank	Space Rank
jade4j	dd47397	<1m	5371	5371	25m	0	-	-	-
jade4j	114e886	<1m	5313	5313	27m	0	-	-	-
HdrHistogram	030aac1	<1m	936	936	3m	7	-	-	-
pdfbox	93c0b69	1m	1375	1375	5m	0	-	-	-
tree-root	fef0f36	<1m	1722	1722	1m	0	-	-	-
spoon	48d3126	9m	0	0	<1m	0	-	-	-
pebble	942aa6e	2m	5104	5104	178m	0	-	-	-
fastjson	c886874	1m	35079	35079	131m	0	-	-	-
htmlelements	bf3f275	1m	3609	3609	9m	0	-	-	-
spring-cloud-connectors	56c6eca	<1m	3258	3258	5m	0	-	-	-
joinmo	a5ee885	<1m	15045	15045	33m	0	-	-	-
buildergenerator	d9d73b3	<1m	750	750	2m	0	-	-	-
mybatis-3	809c35d	7m	28248	28248	119m	0	-	-	-
antlr4	9e7b131	3m	446	446	2m	0	-	-	-
hamcrest-bean	84586d9	<1m	25912	25912	69m	0	-	-	-
raml-java-parser	49aab8f	<1m	1003	1003	3m	0	-	-	-

Table B.15: Results of PAR Templates on CC Defects

Bibliography

[1] AE results. http://dijkstra.cs.virginia.edu/genprog/resources/genprog-ase2013-results.zip.

[2] Bukkit. https://bukkit.org.

[3] CVE-2006-2025. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-2025.

[4] Dataflow java sdk. https://github.com/GoogleCloudPlatform/DataflowJavaSDK.

[5] GenProg benchmarks. http://dijkstra.cs.virginia.edu/genprog/resources/genprog-icse2012-benchmarks/, .

[6] GenProg: Evolutionary Program Repair. http://dijkstra.cs.virginia.edu/genprog/, .

[7] GenProg results. http://dijkstra.cs.virginia.edu/genprog/resources/genprog-icse2012-results.zip, .

[8] GenProg virtual machine. http://dijkstra.cs.virginia.edu/genprog/resources/genprog_images, .

[9] Mining and understanding software enclaves (muse) program. https://wiki.museprogram.org.

[10] Manybugs and introclass benchmarks for automated repair of c programs. http://repairbenchmarks.cs.umass.edu/.

[11] Patch generation via learning (project website). http://groups.csail.mit.edu/pac/patchgen/.

[12] RSRepair results. http://sourceforge.net/projects/rsrepair/files/.

[13] Symantec internet security threat report, Sep. 2006. http://www.symantec.com.

[14] clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[15] dyn.js. http://dynjs.org/.

[16] GitHub. https://github.com/.

[17] Joda-Time. http://www.joda.org/joda-time/.

[18] Junit. http://junit.org/.

[19] Mapstruct - java bean mappings, the easy way! http://mapstruct.org/.

[20] Apache maven. https://maven.apache.org/.

[21] Simple, intelligent, object mapping. http://modelmapper.org/.

[22] OrientDB. http://orientdb.com/orientdb/.

[23] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. doi: 10.1145/503272.503275. URL http://doi.acm.org/10.1145/503272.503275.

[24] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 257–269, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771796. URL http://doi.acm.org/10.1145/2771783.2771796.

[25] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06', pages 158–168. ACM, 2006. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134000. URL http://doi.acm.org/10.1145/1133981.1134000.

[26] Pavol Bielik, Veselin Vechev, and Martin Vechev. Phog: Prababilistic model for code. In *Proceedings of the 33rd International Conference on Machine Learning*, 2016.

[27] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 378–381, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1204-2. doi: 10.1145/2351676.2351752. URL http://doi.acm.org/10.1145/2351676.2351752.

[28] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 609–633. Springer-Verlag, 2011. ISBN 978-3-642-22654-0. URL http://dl.acm.org/citation.cfm?id=2032497.2032537.

[29] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11', pages 121–130, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985811. URL `http://doi.acm.org/10.1145/1985793.1985811`.

[30] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *Computer-Aided Verification (CAV)*, 2016.

[31] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, pages 30–39, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2847-0. doi: 10.1145/2593735.2593740. URL `http://doi.acm.org/10.1145/2593735.2593740`.

[32] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03', pages 78–95, New York, NY, USA, 2003. ACM. ISBN 1-58113-712-5. doi: 10.1145/949305.949314. URL `http://doi.acm.org/10.1145/949305.949314`.

[33] Brian Demsky and Martin C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.

[34] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244, 2006.

[35] Kinga Dobolyi and Westley Weimer. Changing java's semantics for handling null pointer exceptions. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 0:47–56, 2008. ISSN 1071-9458. doi: http://doi.ieeecomputersociety.org/10.1109/ISSRE.2008.59.

[36] Kinga Dobolyi and Westley Weimer. Changing java's semantics for handling null pointer exceptions. In *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*, pages 47–56, 2008.

[37] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. Automatic repair of real bugs: An experience report on the defects4j dataset. *CoRR*, abs/1505.07002, 2015. URL `http://arxiv.org/abs/1505.07002`.

[38] Yong Hun Eom and Brian Demsky. Self-stabilizing java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 287–298, 2012.

[39] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference, GECCO 2010, Proceedings, Portland, Oregon, USA, July 7-11, 2010*, pages 965–972, 2010.

[40] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09', pages 947–954, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-325-9. doi: 10.1145/1569901.1570031. URL `http://doi.acm.org/10.1145/1569901.1570031`.

[41] Zachary P Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

[42] Joel Galenson, Philip Reames, Rastislav Bodík, Björn Hartmann, and Koushik Sen. Codehint: dynamic and interactive synthesis of code snippets. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 653–663, 2014.

[43] 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 *Proceedings of the 37th International Conference on Software Engineering*, 2015.

[44] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. Fixing recurring crash bugs via analyzing Q&A sites. In *Proc. of ASE*, 2015.

[45] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. Deepfix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 1345–1351, 2017. URL `http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603`.

[46] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2015. URL `http://www.gurobi.com`.

[47] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101949. URL `http://doi.acm.org/10.1145/1101908.1101949`.

[48] Manu Jose and Rupak Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11', pages 437–446, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993550. URL http://doi.acm.org/10.1145/1993498.1993550.

[49] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 802–811. IEEE Press, 2013. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486893.

[50] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12', pages 431–450. ACM, 2012. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384648. URL http://doi.acm.org/10.1145/2384616.2384648.

[51] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *Computer-Aided Verification (CAV)*, 2015.

[52] Sebastian Lamelas-Marcote and Martin Monperrus. Automatic Repair of Infinite Loops. Technical Report hal-01144026, University of Lille, 2015. URL https://arxiv.org/pdf/1504.05078.pdf.

[53] Xuan-Bach D. Le, David Lo, and Claire Le Goues. History driven program repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*, pages 213–224, 2016.

[54] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering (to appear)*.

[55] 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 *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 3–13. IEEE Press, 2012. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337225.

[56] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.

[57] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In *Proceedings of the*

fourteenth international conference on Genetic and evolutionary computation conference, pages 959–966. ACM, 2012.

[58] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 298–312.

[59] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786811. URL http://doi.acm.org/10.1145/2786805.2786811.

[60] Fan Long and Martin C. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 702–713, 2016.

[61] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14', pages 227–238, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594337. URL http://doi.acm.org/10.1145/2594291.2594337.

[62] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 727–739, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3106253. URL http://doi.acm.org/10.1145/3106237.3106253.

[63] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 48–61, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065018. URL http://doi.acm.org/10.1145/1065010.1065018.

[64] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Softw. Engg.*, 22(4):1936–1964, August 2017. ISSN 1382-3256. doi: 10.1007/s10664-016-9470-4. URL https://doi.org/10.1007/s10664-016-9470-4.

[65] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the*

38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, pages 691–701, 2016.

[66] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11', pages 329–342, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993537. URL http://doi.acm.org/10.1145/1993498.1993537.

[67] Na Meng, Miryung Kim, and Kathryn S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 502–511, 2013.

[68] Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15', pages 392–402, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818804.

[69] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 234–242, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568324. URL http://doi.acm.org/10.1145/2568225.2568324.

[70] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162, March 2014. doi: 10.1109/ICST.2014.28.

[71] Vijay Nagarajan, Dennis Jeffrey, and Rajiv Gupta. Self-recovery in server programs. In *Proceedings of the 2009 international symposium on Memory management*, pages 49–58. ACM, 2009.

[72] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, August 2011. ISSN 1049-331X. doi: 10.1145/2000791.2000795. URL http://doi.acm.org/10.1145/2000791.2000795.

[73] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13', pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486890.

[74] Huu Hai Nguyen and Martin Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 15–30, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0. doi: 10.1145/1296907.1296912. URL http://doi.acm.org/10.1145/1296907.1296912.

[75] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 2013.

[76] Mike Papadakis and Yves Le Traon. Effective fault localization via mutation analysis: A selective mutation approach. In *ACM Symposium On Applied Computing (SAC'14)*, 2014.

[77] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, page na, 2015. doi: 10.1002/spe.2346. URL https://hal.archives-ouvertes.fr/hal-01078532/document.

[78] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 609–620, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.62. URL https://doi.org/10.1109/ICSE.2017.62.

[79] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Softw. Eng.*, 40(5):427–449, May 2014. ISSN 0098-5589. doi: 10.1109/TSE.2014.2312918. URL http://dx.doi.org/10.1109/TSE.2014.2312918.

[80] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102. ACM, 2009. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629585. URL http://doi.acm.org/10.1145/1629575.1629585.

[81] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, 2013.

[82] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the*

36th International Conference on Software Engineering, ICSE 2014, pages 254–265, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568254. URL http://doi.acm.org/10.1145/2568225.2568254.

[83] Zichao Qi. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. Master's thesis, Massachusetts Institute of Technology. Department of Electrical Engineering and Computer Science., USA, 2015.

[84] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An Analysis of Patch Plausibility and Correctness for Generate-And-Validate Patch Generation Systems (Supplementary Material). http://hdl.handle.net/1721.1/97051.

[85] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An anlysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2015*, 2015.

[86] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14', pages 419–428, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594321. URL http://doi.acm.org/10.1145/2594291.2594321.

[87] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15', pages 111–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677009. URL http://doi.acm.org/10.1145/2676726.2677009.

[88] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16', pages 761–774, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837671. URL http://doi.acm.org/10.1145/2837614.2837671.

[89] Eric Rescorla. Security holes... who cares? In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 6–6, Berkeley, CA, USA, 2003. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251353.1251359.

[90] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[91] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017.

[92] Roopsha Samanta, Oswaldo Olivo, and E. Allen Emerson. Cost-aware automatic program repair. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 268–284, 2014. doi: 10.1007/978-3-319-10936-7_17. URL `http://dx.doi.org/10.1007/978-3-319-10936-7_17`.

[93] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip, and Laurie J. Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 277–287, 2012.

[94] Stelios Sidiroglou, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by multi-application code transfer. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015.

[95] Stelios Sidiroglou-Douskos, Eric Lahtinen, and Martin Rinard. Automatic discovery and patching of buffer and integer overflow errors. Technical Report MIT-CSAIL-TR-2015-018, 2015. URL `http://hdl.handle.net/1721.1/97087`.

[96] Stelios Sidiroglou-Douskos, Eric Lahtinen, Anthony Eden, Fan Long, and Martin Rinard. Codecarboncopy. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 95–105, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3106269. URL `http://doi.acm.org/10.1145/3106237.3106269`.

[97] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 15–26, 2013. doi: 10.1145/2462156.2462195. URL `http://doi.acm.org/10.1145/2462156.2462195`.

[98] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415, 2006. doi: 10.1145/1168857.1168907. URL `http://doi.acm.org/10.1145/1168857.1168907`.

[99] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *Proceedings*

of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011, pages 1069–1084, 2011.

[100] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[101] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 727–738, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950295. URL http://doi.acm.org/10.1145/2950290.2950295.

[102] Nicholas Wang, Michael Fertig, and Sanjay Patel. Y-branches: When you come to a fork in the road, take it. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03', pages 56–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2021-9. URL http://dl.acm.org/citation.cfm?id=942806.943827.

[103] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09', pages 364–374. IEEE Computer Society, 2009. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070536. URL http://dx.doi.org/10.1109/ICSE.2009.5070536.

[104] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE'13*, pages 356–366, 2013.

[105] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, March 2014. ISSN 0018-9529. doi: 10.1109/TR.2013.2285319.

[106] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 416–426, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.45. URL https://doi.org/10.1109/ICSE.2017.45.

[107] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002. ISSN 0098-5589. doi: 10.1109/32.988498. URL http://dx.doi.org/10.1109/32.988498.

[108] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009. URL http://dblp.uni-trier.de/db/journals/ieeesp/ieeesp7.html#ZhivichC09.

[109] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 913–923, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818864.