MUTATION TESTING: ALGORITHMS AND APPLICATIONS

by

David Bingham Brown

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2020

Date of final oral examination: 7/31/20

The dissertation is approved by the following members of the Final Oral Committee:
Thomas Reps, Professor, Computer Sciences
Benjamin Liblit, Amazon. Associate Professor, Computer Sciences (until May 2020)
Somesh Jha, Professor, Computer Sciences
Alexander Stajkovic, Professor, Business

© Copyright by David Bingham Brown 2020
All Rights Reserved

For Laura, who could not be here to read these pages,
but without whom these pages could not exist.

ACKNOWLEDGMENTS

Take me back down where cool water flows
Let me remember things I love

— John Fogerty, "Green River"

Writing in the middle of 2020, it feels like something of a luxury to take the moment to reflect on the recent past to express appreciation for everything that has been done for me during my time in graduate school. This experience has certainly been a challenge, and certainly not in the expected way—many of the things I had expected to be difficult were actually easy, and many of the things I expected to be easy were profoundly difficult. As this adventure draws to a close, I am indebted to many people, and I hope this text manages to recognize a substantial fraction of them.

First I must thank Cassilynn, my ever-patient wife who has always believed in me, even when I have not. I also thank my father for his unwavering support and my children for the patience beyond their years that they have shown.

Critically, I would like to thank my advisors, Thomas Reps and Ben Liblit, for their guidance and more patience than I have deserved over these past six years.

I would like to thank the remainder of my committee: Somesh Jha for being a constant source of difficult yet *always* fair questions, and Alex Stajkovic, for many deeply interesting cross-disciplinary conversations.

I would like to thank the faculty of the Department of Computer Sciences, especially Aws Albarghouthi, Loris D'Antoni, and Bart Miller. I would like to thank the staff of the department as well, especially Angela Thorp, Tae Kidd, and Lance Potter.

I would like to thank my peers in the Department of Computer Sciences, especially my collaborators—Peter Ohmann, Mike Vaughn, and Zi Wang—as well as Ara Vartanian, Jason Breck, Alisa Maas, Jordan Henkel, Venkatesh Srinivasan, and John Cyphert.

I would like to thank Grzegorz Wasilkowski for repeatedly asking the question "Will you consider graduate school?" until the answer was yes. I would like to thank all of my students from the years that I taught software engineering at the University of Kentucky, for being great students and *convincing* me that graduate school was the correct option.

I would like to thank Mirek Truszczyński and Jerzy Jaromczyk for being mentors to me when just beginning my academic career. I would like to thank Jane Hayes, Victor Marek, Ken Calvert, Jim Griffioen, Jeff Ashley, Brent Seales, Hank Dietz, and Jennifer Doerge for being colleagues, advocates, and friends.

I would like to thank Dick Rovinelli for being a mentor and friend during my career as a software engineer.

I would like to thank my friends who have always been supportive and have provided an anchor to sanity during these past several years: Adam Heinz, Brett Strassner, Sam and Marie Osborne, Jake Quimby, Dan Wasmer, Sam Clark, Nathan Hoffmann, Andy Welton, John Cirves, Rick Way, Nicholas Cupery, Don Brandl, Per Nilsson Sandin, Keith and Jackie Christianson, Bradley Given, Travis Marg, Richie Kammer, Will Cruickshanks, Michael Stone, Doug Nordwall, Joshua Cohen, Robin Zebrowski, Jacob Holbird, and Brandon Owens.

This work was supported, in part, by a gift from Rajiv and Ritu Batra; by DARPA under MUSE award FA8750-14-2-0270; by NSF under grants CCF-1217582, CCF-1318489, and CCF-1420866; and and by the UW-Madison OVCRGE with funding from WARF. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

CONTENTS

Contents iv

List of Tables vi

List of Figures vii

Abstract viii

1 Introduction 1
- *1.1 Motivation* 3
- *1.2 Contributions* 6
- *1.3 Thesis Outline* 9

2 Science Communication 10
- *2.1 History (and Prehistory)* 11
- *2.2 Mutation Testing* 14
- *2.3 Improvements to Mutation Testing* 16
- *2.4 Test-Case Generation* 21

3 Wild-Caught Mutants 23
- *3.1 Introduction* 23
- *3.2 Harvesting and Insertion* 27
- *3.3 Experiments* 39
- *3.4 Results* 41
- *3.5 Threats to Validity* 50
- *3.6 Related Work* 52
- *3.7 Experimental Artifacts* 53
- *3.8 Conclusion* 54

4 Guided Mutation Testing 56

4.1 Introduction 56
4.2 Shepherds and Prediction Goals 59
4.3 Training Modalities 65
4.4 Implementation 67
4.5 Experimental Setup and Training 69
4.6 Results 74
4.7 Threats to Validity 80
4.8 Related Work 82
4.9 Conclusion 82

5 Test-Case Generation 84
5.1 Introduction 84
5.2 Motivation 86
5.3 Definitions and Problem Statement 89
5.4 Technique Overview 91
5.5 Target-Oriented Fuzzing 92
5.6 Experimental Setup 96
*5.7 Experiments and Results*100
*5.8 Threats to Validity*110
*5.9 Related Work*111
*5.10 Conclusion*113

6 Conclusion114
*6.1 Mutation Testing*115
*6.2 Future Work*117

References119

LIST OF TABLES

3.1 Most frequent inferred idioms in our experimental corpus 30
3.2 Most frequent identifier shifts in our experimental corpus 38
3.3 Experimental results when harvesting from backward or forward patches 47

4.1 Projects Used 60
4.2 Training Results 70
4.3 Experimental Results on Jannovar ("Held Out") 75

5.1 Mutation-testing frameworks 98
5.2 Test-case generation and TOFU execution times 100
5.3 Test-suite Metrics 105
5.4 Test-suite Metrics 106
5.5 Quality of various proxies for mutant equivalence 107

LIST OF FIGURES

3.1 Overview of mutation-operator extraction and insertion through the mutgen/mutins toolchain . . . 25
3.2 Language-definition file for C. The first character on each line specifies keywords (K), operators (O), quoted string literals (Q), block comments (C), or single-line comments (c). . . . 28
3.3 Example candidate mutation operators. . . . 31
3.4 Examples of real candidate mutation operators found within the experimental corpus. . . . 32
3.5 Potential mutation operators discarded and retained at each filtering stage . . . 36
3.6 Examples of mutation operators proposed by Just et al. (2014b) and identified by mutgen . . . 45
3.6 Examples of mutation operators proposed by Just et al. (2014b) and identified by mutgen . . . 46

4.1 Discernment . . . 64
4.2 Training modalities. . . . 66
4.3 Wild-caught mutants. . . . 68
4.4 Estimated Jannovar mutants filtered by shepherds, discarded by compilation, and retained at each filtering stage . . . 79

5.1 Conceptual Web-Based Interface . . . 88
5.2 Relative Non-equivalence . . . 104

ABSTRACT

Software continues to be vital to the modern world, and as its ubiquity increases, its correctness becomes ever more valuable. Unfortunately, fundamental mathematical constraints on static analysis preclude the possibility of easy answers to the problem of correctness.

Modern software engineering relies upon software testing as an approximate method to establish correctness. Software testing, as exemplified by the use of test suites, provides the approximation desired, but does not present an obvious method to determine the accuracy of the approximation. Mutation testing is a technique that provides both an estimate of the quality of a test suite, as well as an avenue for its improvement.

Mutation testing is an area of active research, and while it is a useful tool for a software engineer, it is not a panacea for the problems of software testing. This dissertation covers research performed with the goals of both improving the state of the art in mutation testing, as well as applying mutation-testing techniques to solve other problems in software engineering.

This dissertation proposes new techniques in the field of mutation testing along three primary lines of research:

1. **"Wild-caught mutants".** We propose a technique for the automatic generation of new mutation operators through the analysis of source-code control repositories.

2. **Guided mutation testing.** We propose a set of techniques to foster prioritization of time-consuming mutation-testing operations.

3. **Automatic construction of test suites.** We propose a technique that uses mutation-testing tools to aid in the process of automatic test-case generation and the automatic construction of entire test suites.

1 INTRODUCTION

Modern civilization relies heavily on software, so much so that there is a recurring joke in the programming-languages-research community that "software is everywhere" slides (or, for that matter, introductory paragraphs) are also everywhere. It is an unfortunate reality that, despite its omnipresence, it is common for software to contain serious flaws. A core focus of the field of software engineering is the detection and eradication of these flaws.

We are, however, constrained by fundamental, mathematical limits on what can be accomplished through analysis itself. The standard solution for defect detection in the software-engineering industry is the test suite. Ultimately, any attempt to describe the behavior of an arbitrary program *must* be an approximation due to the mathematical limits on the accuracy of program analysis, and test suites do not escape these limits. Test suites are necessarily imperfect, but provide an effective way to detect defects introduced in programs as they evolve over time.

Good test suites are among the most important tools available to ensure the quality of software. However, bad test suites help nobody, and evaluating the quality of a test suite itself is challenging. Furthermore, any determination of the quality of a test suite must also, necessarily, be an approximation. Mutation testing provides a method to approximate the quality of a test suite.

Mutation testing of a test suite with respect to a program provides one way to measure the test suite's quality. In essence, mutation testing measures the *adequacy* of the test suite, which is intended to provide an estimate of the ability of the test suite to detect faults inserted into the program in the future (DeMillo et al., 1978; Hamlet, 1977).

The conceptual basis of mutation testing is that one can identify weaknesses in test suites by *mutating* the program under test—that is, modifying the program by making a small, random modification—and executing this newly created *mutant* against the program's test suite. By generating large numbers of mutants, and counting how many of them are detected by the program's test suite, this process measures the quality of a test suite by determining its sensitivity to small changes

in the program.

Several mutation-testing frameworks exist, but most of these frameworks function by randomly modifying the source code of the system under test. These frameworks consist of a set of rules, called *mutation operators*, which are applied to the source code at various points to create mutants. These mutants are then tested against the project's test suite. If the test suite is able to detect the mutant (via a failed test case), the suite is said to have *killed* the mutant; this result provides evidence to the engineer arguing for the robustness of the test suite. The measure of sensitivity referred to earlier is the percentage of mutants that are not killed ("*live* mutants"): the smaller the percentage of live mutants, the better.

Our research aims to provide new techniques in mutation testing to improve both of these core functionalities: the evaluation of test suites, as well as their improvement by the addition of new test cases.

1.1 Motivation

Software engineering is an ever-evolving field, and the central focus of our research is to contribute to its improvement.

The three primary areas of our research are motivated as follows:

Wild-Caught Mutants

Conventional mutation-testing approaches make random (or effectively random) modifications to the target program's code according to some fixed set of substitution directives, such as replacing ">" with ">=". Just et al. (2014b) have shown that this strategy is a useful proxy for real faults.

However, the conventional approach to mutation testing has a basic limitation: the *ad hoc* patterns used do not necessarily reflect the types of changes made to source code by human programmers. Consequently, the measured adequacy does not necessarily reflect how effective the test suite is at identifying the kinds of defects that real programmers might introduce.

By using the revision histories of software projects, we have developed a method for automatically creating new mutation operators that resemble real-world changes made by programmers. We call such mutants *wild-caught mutants*. The objective of our research and development of the "wild-caught-mutants" technique has been to reexamine mutation testing by using mutation operators that more closely resemble defects introduced by real programmers.

Guided Mutation Testing

While it provides useful information about a program's test suite, the value of mutation testing itself is fundamentally limited by its computational cost. On the scale necessary for robust mutation testing, compilation itself is a substantial burden. Once a mutant is created and compiled, the test suite must be executed. Because the time needed to execute the test suite can be arbitrarily high, another computational roadblock is created. This disadvantage can manifest in the unfortunate situation

that without the use of *tremendous* computational resources, the results of an execution of a mutation-testing system can be out of date before completion. That is, the process can take so long that the underlying code has changed significantly through normal development in the time required for the mutation-testing system to operate. This computational barrier significantly limits the wider adoption of mutation testing.

Our aim is to create techniques that improve the real-world applicability of mutation testing by lowering these computational barriers. We propose techniques that employ machine learning to predict successful compilation, semantic equivalence, and the severity of faults introduced through mutation, thereby allowing mutation testing to provide useful feedback while consuming fewer computational resources.

Test-Case Generation

Beyond simply evaluating the quality of test suites, we also seek to improve them. Test-suite development often focuses on code coverage, but this strategy has limits (Inozemtseva and Holmes, 2014). Instead, we attempt to construct stronger test suites through the combination of mutation testing and target-oriented fuzzing. A live mutant provides an example program usable by the engineer to expand the test suite—so long as the mutant program is semantically different from the original program and an input that kills the mutant can be found to create a new test case.

Our goal is to construct both individual test cases, as well as entire test suites. Our strategy to accomplish these goals is to first find live mutants, and then kill them through the application of target-oriented fuzzing. When performed on an individual mutant, a successful application of the technique will yield a test case. When the results of analysis of an entire corpus of mutants are aggregated, the result is a test suite. We believe that our technique can be a powerful tool for an engineer seeking to build a stronger test suite for their project, either by adding individual test cases or generating an entire test suite.

While our ultimate goal is to enable the development of a system for improving

test suites quickly and with relatively low cost, we extend the concept a step further: Beyond just the generation of test cases, we attempt to provide an *argument* for the inclusion of our generated test cases within a project's test suite. We combine a live mutant from the results of mutation testing with an input causing the mutant and original program to exhibit different behavior to justify to the engineer that our constructed test case is *useful*—not only do we create a new test case, but we provide an example fault detected by the new test case.

1.2 Contributions

Ultimately, this thesis is focused around the development of new techniques in mutation testing, both in mutation testing itself as well as its application to the broader field of software engineering.

These contributions are grouped in three broad categories.

Wild-Caught Mutants

We describe a technique—"wild-caught mutants"—that automatically creates mutation operators that others have identified as being missing from previous mutation approaches (Just et al. (2014b)).

We provide a tool-chain for mutation-operator extraction that implements the wild-caught-mutants technique. This tool-chain allows the user to harvest mutation operators from most common programming languages and then apply them to a system under test to perform mutation testing of a test suite.

We report on experiments in which we extracted mutation operators from a corpus consisting of the 50 most-forked C-based projects on GitHub. We found that wild-caught mutants can capture faults that traditional mutation operators are unable to reproduce. Compared to existing mutation operators, the mutation operators obtained by the wild-caught-mutants technique lead to mutants that are roughly as hard to "kill" as mutants from traditional mutation operators. However, they offer a richer variety of changes, and thereby provide a more extensive way to evaluate the quality of a test suite.

Guided Mutation Testing

We describe a set of techniques for training machine-learning models to predict qualities of mutants created during the process of mutation testing. Through the use of models trained with these techniques, an engineer carrying out mutation testing can generate "interesting" mutants for testing a test suite, with the engineer having the ability to specify what particular features they find interesting.

We provide a tool-chain for prioritizing mutants. This tool-chain allows the user to use either "off-the-shelf" or custom-trained models to prioritize mutants generated by a mutation-testing framework. We use our wild-caught-mutants system as the underlying mutator, but other mutation-testing frameworks can be adapted for use with the technique. While one of the features we predict—compilation—is more relevant to our wild-caught-mutants system (as most other systems do not suffer from the low compilation rate of wild-caught mutants), our tools predict other more universally relevant features as well.

We report on experiments in which we analyzed five Java-language projects with voluminous test suites. Models were trained in three different modalities using mutants derived from each project individually and as a group; these models were used to determine both how well each individual model functions on its source project, as well as how well it generalized to other projects. From these data we establish both the cost and benefit of custom-training our models to individual projects.

Test-Case Generation

We describe a technique for creating new and *useful* test cases for a program. We identify potential new test cases via mutation-testing of the program, using target-oriented fuzzing to create a new test case for each live mutant.

We provide a tool-chain for generating these test cases. Our tool-chain allows an engineer to "plug-in" a mutation-testing system to create live mutants, and then generate new test cases by killing these mutants.

We report on experiments in which we generated new test cases for the analyzed program, and constructed several test suites for it. We used this process to determine how well our technique can both generate an individual test case (given a live mutant) and build an entire test suite from a minimal starting point.

Our tool-chain is able to construct both individual test cases, as well as entire test suites. Moreover, it can construct a test suite that has similar performance characteristics to a hand-constructed test suite—as measured by size, code coverage,

and the ability to kill mutants—but at a fraction of the cost in time and dollars.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of my research and its broader context written for an audience not involved in computer-science research. Chapter 3 presents a description of "wild-caught mutants," a new technique for generating mutation operators from the source-control histories of software projects. Chapter 4 presents a discussion of machine-learning algorithms developed to predict qualities of mutants generated through the mutation-testing process. Chapter 5 described the method for automatically generating test cases through a combination of mutation testing and target-oriented fuzzing. Chapter 6 concludes.

2 SCIENCE COMMUNICATION

Some of you may remember the days when computers seemed extremely fragile; the "blue screen of death" is less of a boogeyman today than it was a couple of decades ago, and your phone *rarely* crashes. This change is due in large part to advancements in the field of software engineering. What we may have called supercomputing in the past is now nearly ubiquitous in its current guise of "the cloud." As a civilization we have not only gotten better at *writing* programs, but also (and perhaps more importantly) *understanding* them.

Against this backdrop there may be a certain unusual character to my work to a casual observer. After all, in an era where the most common interaction with a computer is the use of a pocket-sized device with access to the sum of human knowledge, it may seem unwarranted to use what amounts to supercomputing resources to provide incremental improvements to the understanding of individual programs.

My research is within the programming-languages group at the University of Wisconsin–Madison, and more specifically within the area of software engineering. My work has been focused on *mutation testing*—roughly, the idea that one can attain useful knowledge about a program and its associated artifacts by making small, random changes to it and observing the behavior of these "mutants" by executing the program on all elements of a large test suite. This technique is computationally expensive: compiling a large program can be time-consuming, executing the test suite of a robustly-engineered project even more so. The computational expense of the process is great enough that my work would be impossible without access to the high-throughput computing environment available here at the university, and some discussion of *why* this computing power is needed is merited.

To set the stage for what my work *does*, I will—briefly—discuss the history in hopes of explaining how we got here and why there's no simple solution to the problems my work addresses.

2.1 History (and Prehistory)

It is difficult to definitively identify the beginning of software engineering as a field. The concepts of programming existed before the digital computer, and computer scientists discussed issues relating to the correctness of computer programs by at least the 1940's. But it was the 1960's that saw both the coining of the term—in 1965—and the first software-engineering conference—in 1968.

However, thirty years before software engineering existed as a specific concept, critical work in computer science was being done that still resonates to this day.

Alonzo Church and his more-famous graduate student, Alan Turing, performed some of the earliest work in the area of computability—that is, the possibility of computing the answer to a problem or to a class of problems. Church and Turing developed the earliest mathematical models of computation—Church's lambda calculus and Turing's Turing machines—to study this concept.

Much of this critical early work in computer science was focused on two related problems: the *decision problem* and the *halting problem*. The decision problem asks if it is possible for a computer program[1] to determine if *any* formula is universally valid given some set of rules. The halting problem asks if it is possible to construct a computer program P that analyzes *another* program Q and determines whether, for a given input to program Q, execution will terminate. 1936 saw both of these problems answered in the negative—with Church providing the proof that the decision problem cannot be solved in all cases, and Turing providing the proof for the impossibility of the halting problem.

Both of these proofs speak to the *general case*. It is definitely possible to detect *some* valid formulas or equations that always hold, and it is possible to determine if *some* programs will halt (or continue infinitely). The devil in the details is that it cannot be done for *all possible* cases.

These proofs together provide an upper limit to what can be achieved in the area of program analysis. If I may (over-)simplify the ultimate goal of program analysis,

[1]It is worth noting that work on this problem predates modern computers by several years, and the problem was originally phrased in terms of algorithms—that is, a specific list of instructions—as opposed to computer programs, which simply did not yet exist.

it is to determine what a program *does*. We have known for 84 years (at least, as of this writing) that it is impossible for program analysis to determine whether or not a program will ever terminate, given some specific input. Unfortunately, many of the features of a program we wish to ascertain are considerably more complicated—and equally impossible to determine. Because exact answers are impossible, what we are left with is approximations.

If we fast-forward some thirty years after Church and Turing's seminal work, we see an emerging field focused on what was then called the "software crisis." While the difficulties encountered may seem obvious in retrospect, during the early years of software engineering as a field, the scope of the challenges encountered in the authoring and maintenance of software were surprising to many early practitioners. Chief among these challenges was the determination of correctness—especially because the fundamental limits to program analysis were already well known.

A critical heuristic to address the correctness problem was devised. Inputs to a program would be recorded, and the desired output given the input would be recorded as well. This input/output pair is known as a *test case*, and a collection of them is called a *test suite*. While it is mathematically impossible to write a program that determines what another program *does*—at least in the general case—the use of test suites results in an *approximation* of the correctness of the program. That is, if a program satisfies all of its test cases, it is, to some approximation, correct. The failure of specific test cases alerts an engineer to faults in the program, allowing those faults to be detected and corrected more easily.

The common industry practice, called *regression testing*, is to construct and re-run a test suite to test the behavior of a program after it has been modified by an engineer, to ensure that the program is still correct, relative to the test suite. It is unfortunately easy for an engineer working in some area of a program to inadvertently cause a bug to show up in another area. Regression testing makes it possible to identify some of these bugs during the development and testing process. This process is critical to modern software engineering; it is common for the engineers overseeing projects to mandate that a regression test be performed for every change to the program submitted by a developer working on the project.

Today, we call the person responsible for this testing process—and thus, ultimately, responsible for the approximation of program behavior—a software engineer.

2.2 Mutation Testing

Mutation testing was originally proposed by DeMillo, Lipton, and Sayward in 1978 (Demillo et al., 1978). Further work followed in the late 1970's and a tool performing the task first appeared in 1980. Unfortunately the technique has, even from its beginnings, been *extremely* intensive in terms of computational resources, so it would take another twenty years of increases in computational power to see substantial research activity in the field.

Mutation testing as an idea is inseparable from test suites. A test suite, as discussed earlier, represents an approximation of the behavior of a program. We know that it is not a perfect approximation, but we would like to know exactly *how* imperfect of an approximation a particular test suite is. Mutation testing provides one way to obtain such an estimate. It is worth noting that ultimately this method creates an approximation of the quality of an approximation, because any exact measure is mathematically impossible. Such is the cruelty of the halting problem.

This problem—estimating the quality of a test suite—is where mutation testing comes in. The basic idea is that one can evaluate the quality of a test suite by changing the program for which the test suite was constructed and observing if the test suite can identify that the behavior of the changed program is different from that of the original program. We call the program modified in this way a *mutant*, and say that the test suite *kills* the mutant if it is capable of detecting a behavioral difference. Every mutant evaluated in this way provides one of two things to the engineer using the technique:

- If killed, the mutant provides some evidence of the quality of the test suite.
- If live, the mutant provides a modification to the program *not* detected by the test suite, which is an opportunity to improve the test suite by developing a new test case that kills the mutant.

We call the proportion of mutants killed by the test suite the test suite's *mutation-adequacy score*. After performing a full run of mutation testing, the engineer then

receives the mutation-adequacy score of the test suite, as well as a list of mutants *not* killed, and thus suggestions of potential changes to the test suite. These live mutants generated through the mutation-testing process do not *directly* contribute to the test suite, but instead only serve as guidance indicating potential changes undetectable by the suite. My work in test-case generation uses these live mutants as a basis for the expansion of a test suite.

Mutation testing is not without its problems. As mentioned before, it is *extremely* expensive in terms of computational resources. While mutating programs is relatively fast, the compilation process required after the mutant is created can be time-consuming, and executing the test suite on each mutant generated to compute the suite's mutation-adequacy score is typically even more expensive computationally.

Computational costs are not the only problem. It is possible for a mutant to be different syntactically from its parent program—that is, its source code has some difference—but semantically identical—that is, it functions exactly the same as its parent. These mutants are called *equivalent mutants*, and a consequence of the decision problem discussed above is that it is mathematically impossible to catch *all* of them. This limitation creates a set of effective false negatives created by the process. An equivalent mutant will not be caught by a test suite (provided the program being tested is deterministic, at least), and an equivalent mutant presented to an engineer presents no opportunity to improve the test suite because the suite *cannot* be extended to kill the mutant. Thus, equivalent mutants both make a test suite's mutation-adequacy score less accurate *and* waste the time of an engineer reviewing the results of the process.

These problems have limited the adoption of the technique; much of my work has been focused on improving the state of the art of mutation testing.

2.3 Improvements to Mutation Testing

Traditional mutation-testing tools function by making small, random modifications to a program. These modifications are made through the use of a set of potential changes called *mutation operators*. These operators typically make small changes to the program, such as changing the behavior of a comparison (from "greater than" to "less than," for example), or duplicating (or removing) some operation from the program.

One limiting factor of this system is that these changes are, ultimately, arbitrary. Standard mutation operators are chosen by the designer of the mutation-testing tool during the development process of the tool. As a result, while the mutants created with these systems can provide insight into the test suite being evaluated, these changes do not necessarily look like the sorts of changes human programmers make.

My initial research into mutation testing provided a new technique—called "*wild-caught mutants*"—that offers a solution to mutation operators not tracking human-authored changes.

Modern software-engineering projects virtually all use source-control systems. A source-control system is a program that records the history of the source code of a project, collecting all changes to the code, so the engineers using it can revert the project to a specific point in time. This feature is especially useful in finding faults in a program—if the *time* when a fault was introduced to a program can be determined, the source-control system can show the engineers what changes were made at that time, and that information can be used to quickly identify the cause of the fault. The combination of the central repository of source code with the ability to roll back time has proven to be so powerful to software engineers that the use of source control has become ubiquitous within the field.

The wild-caught-mutants technique leverages this history of changes to create new mutation operators. During the *harvesting* phase of the techniques, my tool-chain analyzes all of the recorded changes in a project's history in an attempt to identify small changes that can be turned into mutation operators. These changes

must be relatively small—each typically modifying only a single line of code—and be able to be stored in an abstract form so that they can be applied to other programs. This abstraction mostly functions to allow patterns to be matched in other programs without information specific to each program having to match—the structure of source code is maintained, but features that vary wildly between programs, such as variable names, are treated as wildcards to allow the patterns to match.

Once the harvesting process is completed, the tool-chain emits the set of mutation operators it collected, which can then be used for mutation testing. My tool-chain has been designed from the start to make as few assumptions as possible about the underlying code, and as a result it functions with most programming languages. The harvesting process, while specific to individual languages, can be further customized for individual projects; a user of the system can harvest a set of mutation operators either from publicly available source-control repositories, or by analysis of the history of their own project. In either case, the harvested mutation operators can then be used to create sets of mutants for mutation testing.

I was able to show that this technique was able to create new mutation operators that were not found in existing mutation-testing tools, and do so automatically through the analysis of the source-code repositories of existing projects.

One limitation of the system became apparent during the experimentation—due in large part to design decisions allowing the tool-chain to be largely language-independent, the compilation rate of mutants generated by the system was low compared to traditional mutation-testing tools. My next major project was an attempt to solve this problem, as well as some related issues with mutation testing.

Compilation is the process of converting a program from human-readable (and, really, more importantly: human-writable) source code to a binary format ready to be executed on a computer. While relatively fast for small projects, this step in the build process can consume a large amount of computing resources for large projects, and as a result having a mutation-testing system that generates mutants with a low compilation rate is a problem. To solve this—and some related issues—I developed techniques for *guided mutation testing*.

Guided mutation testing is the idea that mutation testing can be improved

by prioritizing a subset of mutants for more detailed analysis. I approached this problem by using machine-learning techniques to model both the mutation operator, as well as the source code where the mutation operator is applied, and used these models to predict some features of the resulting mutant. My experience with wild-caught mutants made successful compilation an obvious feature to start with, but compilation was not the only prediction made by the system.

My primary goal with guided mutation testing was to reduce the computational power required for mutation testing. As a result, the use of machine-learning techniques may seem counterintuitive, because *training* machine-learning models can be extremely costly in terms of computational resources. While it is definitely true that training is expensive, the *evaluation* of machine-learning models—that is, the computation of a prediction once a model has been fully trained—is very fast, and in the standard use case of the technique, the system will be employed in evaluation mode vastly more frequently than it is trained. Furthermore, while the primary use case is to allow custom-trained models to help a specific software-engineering project, users have the option of using "off-the-shelf," pre-trained models instead, thus requiring no additional resources for training.

Using this technique, I was able to train machine-learning models to predict with high accuracy whether or not a mutant would compile, allowing a user to filter out *most* non-compiling mutants before taking that step. However, that was not the only feature I trained models to predict.

Equivalent mutants are mentioned in the previous chapter as a problem with mutation testing. As a result of the impossibility of the decision problem (discussed in the history of software engineering above), there cannot exist a perfect technique for determining if two mutants are *semantically* equivalent. There are, however, other forms of equivalence that *can* be detected. One of these forms of equivalence is *binary equivalence*.

The compilation process converts human-written source code into computer-understandable machine code. Many elements of source code that allow humans to understand and modify it easily—such as comments, variable or function names, and even formatting of the layout of the text to improve readability—are meaning-

less to a computer and get stripped out by the compiler before the machine code is generated. As a result, is is easily possible for two programs with different source code to have identical machine code. I call two programs *binary equivalent* if their machine code is identical—regardless of the similarity of their source code.

Binary equivalence only captures a subset of semantically equivalent mutants, but this form of equivalence can be determined without error. Notably for the mutation-testing process, analysis of a binary-equivalent mutant is pointless, because it is the exact same program as the original. In addition to the prediction of successful compilation, I was also able to identify most binary-equivalent mutants through a trained machine-learning model.

While mutation-testing fundamentally relies upon the analysis of mutants, not all mutants are equally useful to the process, even after non-compilable and binary equivalent mutants are filtered out. I used machine-learning models to predict one further feature of mutants: *discernment*.

If we view mutation testing as a mechanism both to evaluate the quality of a test suite, as well as provide a basis for its expansion through the identification of its weaknesses, not all mutants are equally valuable. At one extreme, a mutant that causes the program to stop execution essentially immediately after it starts would be so catastrophic that *every* test case in a suite may detect it. Simultaneously, as our detection of equivalent mutants is imperfect, a mutant found by no test cases may be equivalent. Thus the ideal mutant is one that we expect to make a behavioral change (so that it is not an equivalent mutant), but makes a subtle enough change that it is not trivial to detect. *Discernment* is the metric I used to quantify this quality. A discerning mutant is one that is expected to trigger a *small* number of test cases, given a large-enough test suite. That is, create a change in program behavior that is detectable, but not one that is so catastrophic that any functioning test case would be expected to kill it.

I was able to show the ability to predict these three qualities—successful compilation, binary equivalence, and discernment—through use of the guided-mutation-testing technique. Use of the technique allows an engineer to strip down the mutants generated from mutation testing to a small set of mutants more likely to provide

immediate results.

2.4 Test-Case Generation

My work in mutation testing has not been focused entirely on improving the technique. My most-recent work has been a collaboration with other researchers—Zi Wang and Thomas Reps—in the area of test-case generation, producing exciting[2] results. As mentioned earlier, mutation testing provides both an estimate of the quality of a test suite, as well as guidance for the construction of new test cases. My work in test-case generation automates the process of building new test cases from the results of mutation testing.

Test suites are vital to the modern software-engineering process, but they do come with costs. The test cases within these test suites require effort to create, and their execution requires computational resources. Beyond the engineering time spent in the construction of the test suite, computational resources are consumed every time a regression test is performed. High-quality test suites are necessarily large, and eventually can grow large enough that executing them in their entireties can become prohibitively expensive.

My goal with my test-case-generation research is to address both of these issues with software testing. The technique introduced in my work not only allows the generation of new test cases, but also allows a user to generate test cases in such a way as to maximize the utility of the test cases created, so as to avoid adding extraneous cases to the test suite.

If we look at a computer program in its most abstract sense it can be viewed as, effectively, a mathematical function, one that maps some input to some output:

$$output = \mathtt{program}(input) \tag{2.1}$$

As such, a test case can be thought of as the pair $\langle input, output \rangle$, where a program passes the test case if its output when given the provided input matches the test case's recorded output. It follows, then, that it is easy to describe a test-case-

[2]When reading this description, please understand that I am an academic computer scientist, and my definition of "exciting" *may* not line up directly with the definitions in use by all readers of this work.

generation algorithm: choose a random sequence of data as an input, record the program's output, and call this a test case. While this algorithm produces *valid* test cases, the resulting test cases typically evaluate only the program's input validation—that is, the part of the program that (hopefully) detects that its input is garbage and responds appropriately. While testing a program's input-validation routines is valuable, input validation is (typically) only a small part of the program, and thus this algorithm provides limited value when one's goal is testing the *entire* program. Furthermore, every test case generated this way adds more computational costs to *every* regression test conducted. This additional cost necessitates test cases to have some level of quality; because every test case added to the suite imposes computational costs in the future, test cases have to provide an amount of value to the engineer that justifies not only the cost of their creation, but the cost of their execution over time.

So, more *interesting* test cases are needed. My work here builds upon mutation testing in the hopes of generating test cases that meet a basic threshold of interest.

3 WILD-CAUGHT MUTANTS

3.1 Introduction

Good test suites are among the most important tools available to ensure the quality of software. However, bad test suites help nobody, and evaluating test suites themselves is challenging. Mutation testing of a test suite with respect to a program provides one way to measure the test suite's quality. In essence, mutation testing measures the *sensitivity* of the test suite, which is intended to provide an estimate of the ability of the test suite to detect faults inserted into the program in the future (DeMillo et al., 1978; Hamlet, 1977). Conventional mutation-testing approaches make random (or effectively random) modifications to the target program's code according to some fixed set of substitution directives, such as replacing ">" with ">=". Just et al. (2014b) have shown that this strategy is a useful proxy for real faults.

However, the conventional approach to mutation testing has a basic limitation: the *ad hoc* patterns used do not necessarily reflect the types of changes made to source code by human programmers. Consequently, the measured sensitivity does not necessarily reflect how effective the test suite is at identifying the kinds of defects that real programmers might introduce.

The objective of our research has been to reexamine mutation testing by using mutation operators that more closely resemble defects introduced by real programmers. Thus, the high-level goal of our work is as follows:

> Find a method for creating potential faults that are closely coupled with defects created by actual programmers.

We have developed a method for identifying such mutation operators by using the revision histories of software projects. We call such mutants *wild-caught mutants*.

When interpreting a revision history, it may be difficult to determine precisely when a defect was introduced. For this reason, we use instead the *reversal*

of what is likely to be a *correction* in the revision history. That is, the orientation of a mutation operator that we recover is *backward* with respect to the direction of the patch from which it was recovered in the revision history. From a patch of the form "before-code $\rightarrow$ after-code," we create a mutation operator "$\text{pattern}_A \Rightarrow \text{replacement}_B$," where pattern_A is a pattern created from code fragment "after-code," and replacement_B is a rewrite created from code fragment "before-code."[1]

After we present the details of our method for extracting such mutation operators, there are a number of natural research questions that we consider. For starters, we wish to know whether wild-caught mutation operators subsume the manually curated mutation operators widely used until now:

> Research Question 1: Does the the operator-harvesting method of the wild-caught-mutants technique find existing mutation operators?

Conversely, perhaps wild-caught mutants exhibit useful qualities that go beyond past work:

> Research Question 2: Does the operator-harvesting method of the wild-caught-mutants technique find operators that are *not* existing mutation operators?

We also want to know whether our approach leads to improved mutation testing:

> Research Question 3: Do wild-caught mutants exhibit behavior that is quantifiably different than existing mutation operators—and, if so, in what ways?

While backward patches seem more likely to (re)introduce bugs, which is good from the standpoint of mutation testing, forward patches may also describe interesting human-generated changes.

[1]While a patch "before-code $\rightarrow$ after-code" could introduce a defect—and hence our recovered mutation operator would represent a correction—our system confines itself to small and typically single-line patches, which one might expect to be corrections more often than defect introductions.

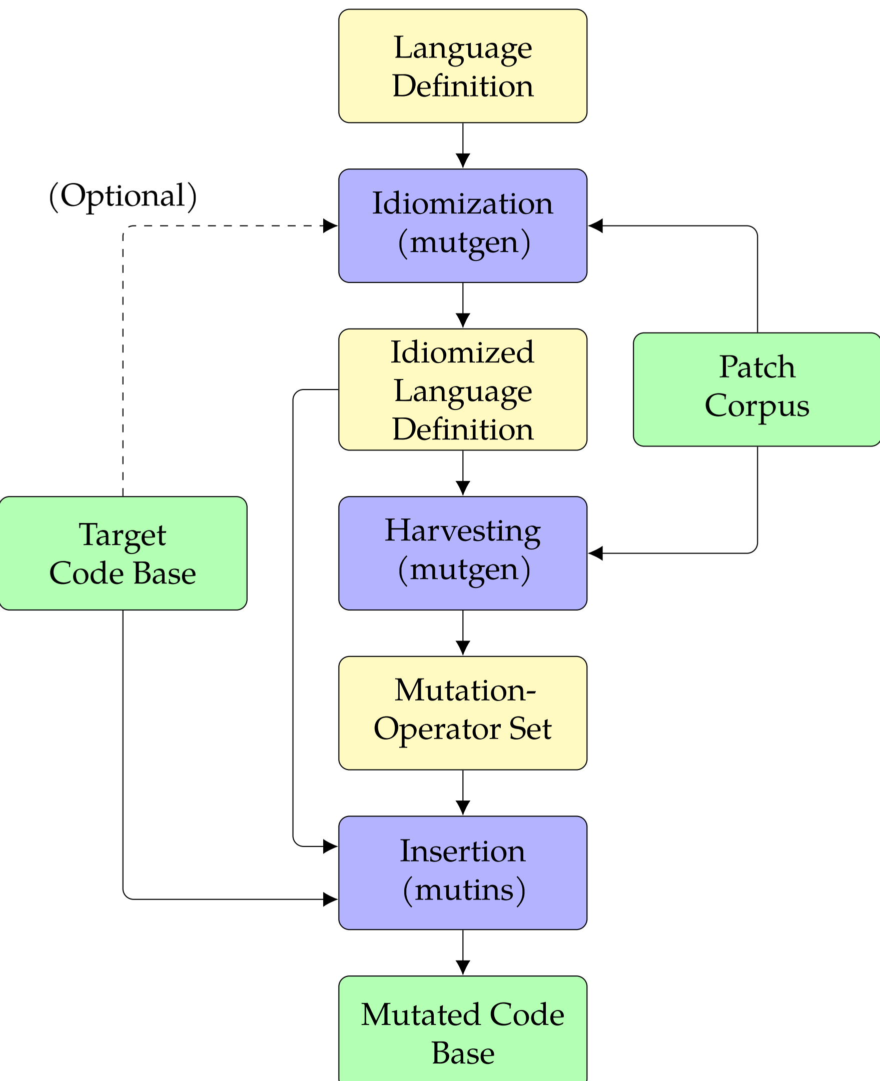


Figure 3.1: Overview of mutation-operator extraction and insertion through the mutgen/mutins toolchain

Research Question 4: Does harvesting from forward patches yield many additional mutation operators, and do the behaviors of these new operators differ significantly from those harvested from backward patches?

The contributions of our work can be summarized as follows:

We describe a technique to automatically create mutation operators that others have identified as being missing from previous mutation approaches (Just et al. (2014b)).

We created a toolchain for mutation-operator extraction that implements the wild-caught-mutants technique. This toolchain allows the user to harvest mutation operators from most common programming languages (using mutgen) and then apply them to a system (using mutins) to perform mutation testing of a test suite.

We report on experiments in which we extracted mutation operators from a corpus consisting of the 50 most-forked C-based projects on GitHub. We find that wild-caught mutants can capture faults that traditional mutation operators are unable to reproduce. Compared to existing mutation operators, the mutation operators obtained by the wild-caught-mutants technique lead to mutants that roughly as hard to "kill" as mutants from traditional mutation operators. However, they offer a richer variety of changes, and thereby provide a more extensive way to evaluate the quality of a test suite. Harvesting from forward patches provides a significant number of operators not obtained from backward patches. There is some support for the conjecture that, compared to forward-harvested operators, backward-harvested operators can introduce defects that more closely resemble defects introduced by real programmers.

Organization. The remainder of the chapter is organized as follows: Section 3.2 offers an overview of our approach, then describes mutation-operator extraction and mutant insertion in detail. Section 3.3 presents our experimental setup, followed by experimental results in section 5.7. Section 3.5 considers threats to the validity of our approach. Section 3.6 discusses related work. Section 3.7 describes supporting

materials that are intended to help others build on our work. Section 5.10 concludes.

3.2 Harvesting and Insertion

As shown in fig. 3.1, our system consists of two tools: mutgen, for extracting reusable mutation operators, and mutins, for applying these operators to the code of a system under test. The extraction process—referred to as *harvesting*—generates a *mutation-operator set* from a corpus of diff-formatted code patches. Our insertion tool, mutins, can then apply these mutation operators to a new code base distinct from that used during harvesting. The main input to mutgen is a corpus of patches for harvesting; the main input to mutins is a target code base to mutate. Both tools are parameterized by a second input, a *language definition*, which specifies the syntactic elements of the language on which they operate (see section 3.2). In other words, our implementation of the wild-caught-mutants approach is really a framework that can be retargeted easily to work on other languages.

Our toolchain operates in three phases: idiomization, harvesting, and insertion. *Idiomization* augments the language definition to conform more closely with the patch corpus or system under test. This preprocessing step is performed by mutgen and is described in section 3.2. *Harvesting* extracts novel mutation operators from the patch corpus. This process is also performed by mutgen and is described in section 3.2. The final *insertion* step applies harvested mutation operators to the system under test. It is implemented by mutins and described in section 3.2.

Language Definition

We define a language as a set of operators, keywords, quote delimiters, and comments (both block comments and single-line comments). Our language parser is essentially a lexical analyzer. We chose to utilize a strictly lexical—as opposed to AST-based or similar—as to allow our system to be more flexible with regard to its inputs. A purely lexical model does not require the harvester to be able to compile any part of its input corpus and can operate on code fragments without additional

```
K auto break case char const continue default
K do double else enum extern float for goto
K if inline int long register restrict return
K short signed sizeof static struct switch
K typedef union unsigned void volatile while
O = += -= *= /= %= &= |= ^= <<= >>= ++ --
O + - * / % ~ & | ^ << >> ! && ||
O == != < > <= >= [ ] -> . ( ) , ? :
Q ' "
C /* */
c //
```

Figure 3.2: Language-definition file for C. The first character on each line specifies keywords (K), operators (O), quoted string literals (Q), block comments (C), or single-line comments (c).

context. However, our language-definition files are simpler than those required by a full lexical-analyzer generator (e.g., flex), in part because we do not need to feed tokens into a full compiler. Additional simplification is possible by leveraging commonalities seen in the basic syntax of C and other C-influenced languages, such as C++, Java, and C#. These commonalities allow us to recognize tokens with high accuracy using the following strategy:

- Operators are consumed greedily from the input stream.
- Text from one quote delimiter to a matching quote delimiter is parsed as a single string literal.[2]
- Text identified to be not part of an operator is read until whitespace or an operator is encountered; once isolated in this fashion, the text is classified as follows:
  - If the text exists in the keyword list, it is classified as a keyword.
  - If the text begins with a digit, it is classified as a numeric literal.

[2]Escaped quote delimiters are not handled, but would not be hard to add.

– Otherwise, it is classified as an identifier.

While simple, these rules are such that mutgen can effectively parse most languages that lack semantic whitespace.[3]

Using such a simple language definition—and not, e.g., a context-free grammar for the language used in the corpus—supports our goal of using patch histories as a source of mutation-operator sets. The patches processed by mutgen to harvest mutation-operator sets have varying and unpredictable contexts: one may easily encounter a patch that begins or ends mid-expression or mid-comment. Therefore, we do not parse the input with respect to a context-free grammar for the language of the corpus. Instead, we perform purely lexical analysis, generating a stream of tokens as determined by rules in the language definition. This approach also allows our system to be more flexible with regard to its inputs. The harvester need not be able to compile any part of its input corpus; it can injest incomplete or invalid code fragments as well as complete code.

Comments create a challenge during harvesting. The patches we process can begin and/or end mid-comment. Thus, we cannot guarantee that the tokens generated when analyzing a particular patch correspond to actual code, as opposed to a natural-language comment. We address this problem in both the extraction and insertion phases. During extraction, we use heuristics to identify (and discard) patches that are likely to be comments; section 3.2 discusses these heuristics in more detail. During insertion, we have the full system under test—and therefore the complete context for any potential insertion—so we can identify comments precisely and exclude them from mutant insertion.

Idiomization

The language-definition file covers all of a language's keywords and operators. However, some identifiers are used so often, and in such standardized ways, as to effectively be additional keywords. We call these identifiers *idioms* and the process

[3]Python's semantically-meaningful whitespace could be handled as well by materializing and later dissolving explicit indent/outdent tokens: a standard Python lexing/parsing technique.

Table 3.1: Most frequent inferred idioms in our experimental corpus

Idiom	Incidence
0	1 350 306
0x00	984 949
dev	695 548
1	578 986
set	491 888

Idiom	Incidence
2	489 152
u32	386 045
y	379 709
u8	311 762
file	310 500

Idiom	Incidence
s	294 490
line	227 925
data	227 376
inode	216 908
o	215 515

of identifying them *idiomization*. We collect identifiers that occur within the corpus above a user-selected threshold. This threshold can be specified by minimum incidence in the corpus, minimum frequency, or a "top-k" limit of accepted idioms. Subsequent extraction passes treat these identifiers as additional keywords.

For example, NULL is missing from the C language definition given in fig. 3.2. This omission is correct: NULL is a standard C macro but is not a C keyword *per se*. Adding NULL to this definition by hand would be easy, and would expand the pool of admissible candidate mutation operators. Unfortunately, the arbitrary choices required by this method do not necessarily scale. Automated idiomization provides a pragmatic method to augment a base language definition with identifiers that are used idiomatically in practice.

Depending on differences in the subject of the patch corpus and the system under test, idiomization has the potential to identify idiomatic keywords that appear rarely or never in the system under test. Therefore, we make idiomization optional, and also allow the system under test to be used as its own source of idiomization.

Table 3.1 lists some of the most commonly identified idioms derived from our experimental corpus. The influence of the Linux kernel is apparent in several entries.

Syntactic Mutation

Mutgen identifies candidate mutation operators by isolating small changes (defined as having fewer than a configurable number of lexical tokens) from the revision

```
− if (x)
+ if (x && y)
```

(a) Admissible candidate

```
− :if .( $1 .)
+ :if .( $1 .&& $_ .)
```

(b) Mutation operator extracted from fig. 3.3a

```
− if (x && y)
+ if (x)
```

(c) Inadmissible candidate: requires synthesizing “y”

```
− if (x > 0)
+ if (x > 1)
```

(d) Candidate made admissible by idiomization of “0”

Figure 3.3: Example candidate mutation operators.

history it reads as input. For a patch to be considered for extraction, it must contain a contiguous section of removed and replaced code that we divide into a “before” and “after” block. A single patch (that is, a single diff-formatted file) can contain multiple blocks of modified code, and each individual contiguous block is treated as a separate candidate mutation operator.

Corresponding blocks of each identified section are broken into a stream of tokens as described by the language definition (see section 3.2). Mutgen makes no attempt to understand the underlying semantics or grammar of a processed language.

Mutins does not attempt synthesis of identifiers or literals, so mutgen requires that candidate mutation operators not require the synthesis of new information. In particular, it must be possible to assemble the *before* state solely from identifiers and literals matched in the *after* state, along with any keywords drawn from the idiomization-enhanced language definition. Once the *before* and *after* blocks are tokenized, mutgen then analyzes both to determine whether this requirement is satisfied. A candidate mutant that meets the requirement is called an *admissible candidate*. Figure 3.3a shows an admissible candidate mutation operator: building the *before* text requires no new identifiers or literals beyond those that appeared in the corresponding *after* text.

Conversely, an *inadmissible* candidate is one that would require synthesis of new information to turn its *after* state back into its *before* state. The candidate mutation operator in fig. 3.3c would be discarded as inadmissible: its *before* state includes the identifier "y," which is not found anywhere in the *after* state.

The idiomization process discussed above allows for limited synthesis of terms not present in the *after* state. Thus, idiomization turns some otherwise-inadmissible candidates into admissible ones. The candidate mutation operator in fig. 3.3d would be inadmissible if we had to synthesize the "0" in the *before* state. However, "0" is so common that it is always recognized as an idiomatic keyword in practice. Thus, the *before* state of fig. 3.3d can be constructed from the *after* state by replacing "1" with the idiomatic keyword "0".

Figure 3.3b shows the tokenized mutation operator extracted and generalized from fig. 3.3a. In the mutation-operator language of mutgen and mutins, ":" indicates a keyword, where the text that follows identifies the keyword itself. Likewise, "." indicates an operator, where the text that follows specifies the operator text. For identifiers and literals that appear in both the *before* and *after* states, we number *after* identifiers and literals starting from 1. "$i" represents the i^{th} identifier or literal. This notation lets us represent mutation operators that are polymorphic with respect to identifier names and literal values. Thus, the generalized mutation operator in fig. 3.3b can match a wide variety of "if" statements, not merely those that test the value of "x && y," as in fig. 3.3a. "$_" marks identifiers and literals that do not appear in the *before* text.

As seen in fig. 3.3b, mutation operators are stored in a plain-text format that humans can easily read and edit. This feature allows a user to create hand-written mutation operators for use in our mutation-testing system.

Figure 3.4 shows some candidate mutation operators identified by mutgen. Figure 3.4a shows a patch that fixes a missing else keyword. From this patch, we harvest a mutation operator that can remove any else keyword immediately after a right curly bracket. The patch in fig. 3.4b generalizes into a mutation operator that can add a semi-colon to certain while statements. Figure 3.4c yields a mutation operator that, when applied to C code, can strip a modulo operation applied to a

```
– }                     – while (i < n);          – TMPFILE
+ } else                + while (i < n)           + TMPFILE % 512
```

(a) (b) (c)

Figure 3.4: Examples of real candidate mutation operators found within the experimental corpus.

single identifier.

Filtering Heuristics

Our initial expectation that admissible candidate mutation operators would be rare proved to be untrue. On the contrary, our initial run of mutgen over the patch corpus used in our experiments yielded over twenty million mutation operators: more than it was reasonably possible to evaluate. Manual inspection revealed that many of these were not worth keeping. For example, some would only mutate comments or were so complex as to be unlikely to match token sequences from other code bases. We therefore extended mutgen with heuristic filters to detect and discard operators with less-promising potential. Mutgen's complete filtering sequence, applied in the order given below, is as follows:

1. **Too many tokens:** Candidates that consist of large amounts of code are so specific that they will probably not match in any other code base. Therefore, we discard candidates that affect eleven or more tokens.

2. **Too few tokens:** Conversely, single-token candidates would match too frequently to be practical. Therefore, we require that either the *before* or the *after* text contain at least two tokens. Note that identifier shifts (section 3.2) can still apply to single-token *before* and *after* texts.

3. **ASCII art:** Candidates that contain three or more repeated operators are assumed to be comments and excluded. This situation commonly arises in line-spanning ASCII art such as "****," "––––," or "////."

4. **Comment detected:** Using the language definition (section 3.2), we can recognize candidates that include the start or end of a comment. Mutating comments is not useful in mutation testing, so we exclude such changes.

5. **Needs synthesis:** Per section 3.2, we discard candidates for which the *pattern* contains identifiers or literals not present in the *replacement*. Our current version of the toolchain does not support the synthesis of identifiers (outside of the limited identifier conversion in identifier shifts or through the use of identifiers treated as keywords through idiomization) that would be required to apply these mutation operators to a target program.

6. **Too many identifiers:** As for "Too many tokens," a candidate involving too many identifiers is unlikely to be applicable to new code. We discard candidates that affect more than four identifiers.

7. **Too many adjacent identifiers:** A candidate containing three "identifiers" in a row is more likely to be natural-language text than programming-language source code; we assume that these candidates involve comments and exclude them from harvesting.

8. **Identical tokenized strings:** Generalizing a candidate into a reusable mutation operator can make the before and after token streams identical, yielding a "mutation" operator that changes nothing. This situation can arise, for example, when the candidate merely affects whitespace. We discard these candidates.

9. **Unbalanced brackets:** All mainstream languages include bracketing tokens that must appear in matched pairs, such as round parentheses, square brackets, and curly braces. Mutation operators can introduce mismatches when our harvester splits one commit into multiple separate changes, each of which affects only one side of a matched-bracket pair. We discard candidates that introduce mismatched counts of opening and closing round parentheses, square brackets, or curly braces. Introducing this filter increased the compi-

lation rate of mutants produced by the toolchain from about 8.7% to about 14%.

10. **Duplicated mutation operator:** Similar candidates can yield identical generalized mutation operators. We discard redundant copies.

Several of the above filters rely on configurable thresholds. Adjusting thresholds for total tokens or potential identifiers can dramatically change the total harvested mutation operators, while also changing the probability that each mutation operator can be matched with (and therefore inserted into) another code base. Some tuning may be needed for specific languages or coding styles. For example, function application can lead to multiple adjacent identifiers in many functional languages: "sum x y z" in ML or "(sum x y z)" in Lisp instead of "sum(x, y, z)" in C. In such languages, the "Too many adjacent identifiers" filter should only be used with a high threshold.

The result of applying these culling heuristics was to reduce the generated set from over twenty million mutation operators, most of them unusable, to roughly forty four thousand, of which a larger proportion can be applied to other code bases. Section 3.2 discusses the empirical behavior of these filters in greater detail.

Effect of Filtering Heuristics

Figure 4.4 depicts the filtering process as applied to Space. Flow begins at the top with 20 063 907 candidates and proceeds downward. Each filter removes some candidates and allows others to proceed to later stages. The width of each curved arrow represents the absolute number of potential mutation operators discarded at each stage; the actual count is reported immediately after the colon in each stage's description. For example, "Duplicated mutation operator" discards 219 569 candidates. The diminishing width of the straight flow descending along the right edge of the diagram is proportional to the number of candidates retained after all preceding steps. Numbers in parentheses are the fraction of surviving candidates discarded, expressed as percentage of candidates considered at each stage, not as a

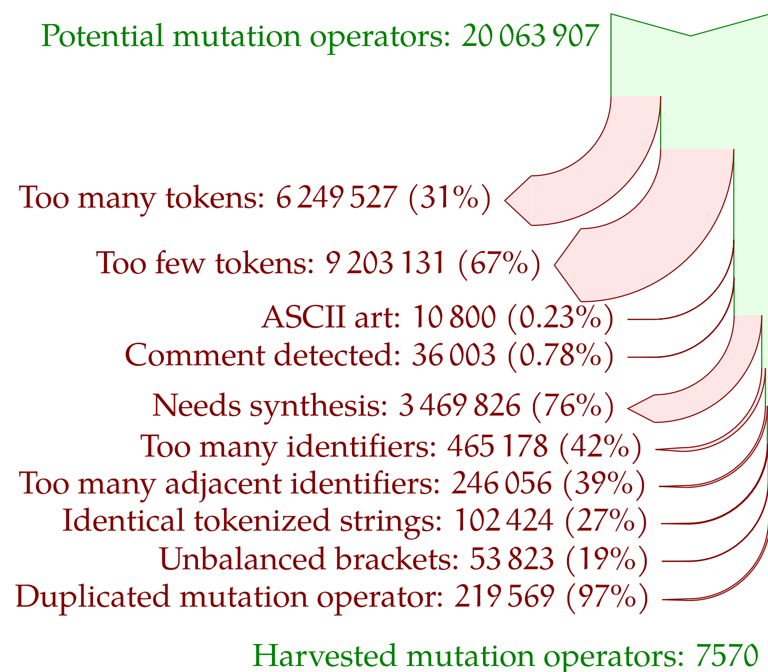


Figure 3.5: Potential mutation operators discarded and retained at each filtering stage

percentage of the 20 063 907 potential mutation operators gathered at the start. For example, the "Duplicated mutation operator" filter discards 97% of the mutation operators that had not already been eliminated in any preceding stage.

In absolute terms, "Too few tokens" is the major gatekeeper, accounting for nearly half of the initial candidates that do not make it through to the end. "Too many tokens" and "Needs synthesis" also discard large portions of the initial pool. The latter could potentially be relaxed by deeper semantic analysis to allow more ambitious synthesis beyond our idiomization technique. The other filters seem minor relative to the large starting candidate pool, but notice that each of these still discards tens or hundreds of thousands of candidates. "ASCII-art" detection is highly selective and therefore has the least impact, discarding just 0.23% of the potential mutation operators it considers, but even this filter eliminates 10 800 candidates that would have been pointless to turn into mutation operators. When

operating at these large scales, even relatively small contributors can be important.

Identifier Shifts

We call the second type of wild-caught mutant extracted by mutgen an *identifier shift*. During the extraction process, it is common for mutgen to identify a patch that consists solely of a change of one identifier to another. While the syntactic-mutation technique explicitly avoids synthesis during the insertion process (with the exception of a limited form permitted through idiomization), we capture these single-identifier changes, calling them *identifier shifts*, to allow an additional, limited form of synthesis.

Any patch that is observed to replace *solely* one identifier with another is marked as a candidate identifier shift. At the end of extraction, all candidate shifts with incidence above a configurable threshold are encoded as identifier shifts within the mutation-operator set. All identifier shifts extracted during the harvesting process are used both "forwards" and "backwards." That is, a single identifier-shift mutation operator can replace either the "before" identifier with the "after" or vice versa.

Table 3.2 shows example identifier shifts harvested from the corpus used in our experiments.

Insertion

Once the harvesting process produces a mutation-operator set, our mutant-insertion tool mutins can then apply mutation operators to a code base.

Mutins works by tokenizing all source-code input files using the same language-definition specification and rules discussed in section 3.2. It then selects a mutation operator from the mutation-operator set. By default, the selection is done randomly, but the user may specify either a particular mutation operator in the mutation-operator set by index, or specify a seed for the random-number generator.[4]

[4]Mutins uses the Mersenne Twister(Matsumoto and Nishimura, 1998) random-number generator, both for the generation of high-quality random numbers, as well as to allow seeds to be used

Table 3.2: Most frequent identifier shifts in our experimental corpus

Before	After	Incidence
__init	__devinit	12 967
module_exit	module_platform_driver	12 897
DEVICE_PRT	DBG_PRT	10 097
of_device	platform_device	8704
m	y	6912
CONFIG_PM	CONFIG_PM_SLEEP	6617
CONFIG_EMBEDDED	CONFIG_EXPERT	6148
mach	plat	5963
A_UINT8	u8	5658
device	platform_device	5610

Once the mutation operator is selected, mutins then attempts to match the mutation operator's pattern to any subset of the token stream generated from parsing the source code. All possible matches are identified, and if any exist, one is chosen randomly if the *insertion index* is not specified by the user. Mutins then replaces the tokens in the source file—preserving whitespace—with the tokens from the replacement in the mutation operator.

To avoid inserting mutants into non-executable portions of the source code, mutins uses the comment rules defined in the language-definition file—see fig. 3.2—to identify comments during the insertion process, and does not apply mutation operators to token sequences that lie within comments. In contrast to the harvesting process, the mutation-insertion process has the entire source file available for analysis, and so can more reliably identify comments because the full context is visible.

---

across systems and allow faithful reproduction of random sequences.

3.3 Experiments

Repository Mining

We obtained mutation operators by mining public GitHub repositories that contain C code. We wanted to target the repositories with the largest number of commits; however, the GitHub API does not provide a way to search based on the number of commits. As a proxy for number of commits, we opted instead to select those repositories with the most *forks*, which is accessible via GitHub's API. The number of forks would seem to be a reasonable heuristic for projects with significant activity—and thus a higher rate of development, and commits from more developers. Qualitatively, the assumption appears to be warranted: the top 20 project repositories under this metric include the Linux kernel (Torvals, 2017), memcached (memcached community, 2017), and Redis (Sanfilippo, 2017). For our experiments, we used the full revision histories of the top 50 project repositories, which consisted of approximately 600 thousand commits containing roughly 20 million individual diff blocks spanning 850 million lines of text.

Target Program

We used the 50 project histories to rerun (part of) an experiment reported by Andrews et al. (2005a), substituting the wild-caught mutation operators obtained from the 50 GitHub project histories for the set of mutation operators used by Andrews et al.

Andrews et al. experimented on programs from the SIR repository (Do et al., 2005). For each program, Andrews et al. generated a number of test suites by randomly choosing a subset of the tests in the program's full test suite. They then measured the mutation adequacy of each randomly chosen test suite by running each test suite over the set of all mutants of the program created by applying a single mutation operator at a single site in the program. By collecting these measurements, Andrews et al. constructed a model of the statistical distribution of the mutant-detection rate over arbitrary test suites, which they compared to a

similarly constructed approximation of the distribution of hand-seeded faults.

To test the effectiveness of mutation testing, Andrews et al. worked with a wide variety of programs from SIR, including the Siemens suite. Among these, Space (Vokolos and Frankl, 1998; Wong et al., 1999) was the only program that they tested for which *real* faults were available instead of hand-introduced ones. Because we were interested in understanding how wild-caught mutants fare against a test suite's detection rates for real faults, we worked only with Space. As distributed by SIR, Space has 38 buggy variants and one "gold" version with no known faults. Andrews et al. used the bug-free gold version; we did the same to allow direct comparison with Andrews et al.'s findings.

Procedure

Following the method of Andrews et al., we generated 5000 100-case test-suite subsets from Space's set of 13 496 total test cases. Next, we ran mutins on Space, to identify each possible point at which a wild-caught mutation operator can be applied. We recorded each possible insertion in a list that could be fed to our test-suite-execution framework at a later time. We then divided the space of mutation insertion points into batches to be run in parallel on a large-scale computing platform capable of serving over 300 million hours of compute time annually.

We inserted each mutation and compiled the result; if compilation succeeded, we ran each of the 5000 100-case test-suite subsets. The data was gathered in parallel because there are no interdependences among any of the runs of a test-suite subset.

Once all test batches completed, we recorded the number of mutants that successfully compiled. We also computed the *mutation-detection ratio*, $\mathrm{Am}(S)$, for each compiled mutant and each test suite, defined as follows:

Definition 3.1. *Let S be a test suite. Then the* mutation-detection ratio $Am(S)$ *is defined as follows:*

$$Am(S) = \frac{\#\ \textit{of mutants detected by } S}{\#\ \textit{of mutants not equivalent to the original program}}.$$

The denominator of Am(S) requires determining whether each mutant is equivalent to the original program, which is undecidable in general (Budd and Angluin, 1982; DeMillo et al., 1978; Offutt and Pan, 1996). Therefore, Andrews et al. (2005a) adopt, and we reuse here, a decidable approximation:

$$\text{Am}(S) = \frac{\text{\# of mutants detected by S}}{\text{\# of mutants detected by program's complete test suite}}.$$

In other words, any mutant that triggers no failure in Space's extensive 13 496-case complete test suite is assumed to be equivalent.

3.4 Results

Research Question 1: Do Wild-Caught Mutation Operators Cover Existing Mutation Operators?

Just et al. (2014b) describe a set of mutation operators provided by the Major mutation framework (Just, 2014):

- **Replace constants.** Mutgen can extract mutation operators that replace constants in the system under test both through the idiomization technique (effectively turning literal constants into language keywords, which can then be extracted in the form of a syntactic mutation operator) or identifier shifts. If specific conversions are not found within the corpus from which mutation operators are harvested, a mutins user can manually add mutation operators that replace specific constants.

- **Replace operators.** All operators seen in the language-definition file used as an input to mutgen are capable of being extracted as syntactic mutation operator. Operator replacements can also be added manually to the mutation-operator set.

- **Modify branch conditions.** Operators to modify branch conditions can be extracted as syntactic mutation operators. (section 3.2).
- **Delete statements.** Mutgen does not yet support the harvesting of statement deletions, but there is no impediment to doing so. In the terminology of Section 5.1, from a patch of the form

$$\epsilon \rightarrow \text{after-code},$$

  we can create a statement-deletion operator of the form

$$\text{pattern}_A \Rightarrow \epsilon.$$

Our framework allows for the replication of all four classes of mutation operators, although mutgen does not currently harvest statement-deletion operators. Future work will support the harvesting of these mutation operators; mutins already supports such mutation operators if mutgen were capable of producing them.

The PIT Mutation Testing suite(Coles, 2017) supports a set of eleven non-experimental mutation operators, many of which duplicate mutation operators provided by the Major mutation framework:

- **Conditionals Boundary Mutator, Conditionals Mutator, Invert Negatives Mutator, Math Mutator, Negate Increments Mutator.** Mutins can replicate these mutation operators in the same manner as Major's **Replace operators** mutation operators, as all of these mutation operators consist of replacing individual operators (or omit a unary minus from a larger expression, in the case of **Invert Negatives**).
- **Return Values Mutator.** Mutins can duplicate this mutation operator via idiomization (to harvest common literal numeric values 0 and 1) or via syntactic mutation operators for the language-specific keywords true, false, and null[5].

[5]The PIT framework operates on Java; while these keywords do not exist in the C language definition used in the experimentation in this chapter, a Java language definition for mutgen prop-

- **Void Method Calls Mutator** Mutins does not currently support statement deletion, of which this mutation operator is an instance.
- **Inline Constant Mutator.** Mutins can replicate this mutation operator through idiomization as in Major's **Replace constants** mutation operator.
- **Remove Conditionals Mutator, Constructor Calls Mutator, Non Void Method Calls Mutator.** Mutins can utilize harvested mutation operators of these types, so long as an example of a change of the type exists within the input corpus.

Our framework allows for the replication of ten out of the eleven non-experimental mutation operators supplied by PIT, again failing to directly reproduce statement deletion.

Research Question 2: Do Wild-Caught Mutation Operators Extend Existing Mutation Operators?

In their study of whether real faults are coupled to mutants, Just et al. (2014b) found that for 27% of the real faults in their study, none of the triggering tests detected any additional mutants. They manually reviewed those faults, and classified them as follows: (i) cases where a mutation operator should be strengthened; (ii) cases where a new mutation operator should be introduced; and (iii) cases where no obvious mutation operator can generate mutants that are coupled to the real fault

In our experiments, we found that several of the mutation operators identified by Just et al. appeared among the mutation operators harvested by mutgen. Specifically, we are able to harvest mutation operators that are consistent with the classifications of Just et al.:

Stronger mutation operators

erly identifies them as keywords and treats them as such during the harvesting process without idiomization.

- *Argument swapping.* Mutgen is capable of harvesting patches that rearrange function arguments, which become mutation operators that perform the inverse rearrangement.

- *Argument omission.* Mutgen is capable of harvesting patches that contain a function call modified to have additional arguments, which become mutation operators that match a function call and replace it with one that has fewer arguments.

- *Similar library method called.* The identifier-shift technique (Section 3.2) allows mutgen to harvest mutation operators of this category by identifying patches in which a single identifier is replaced by another.

  Just et al. specifically mention a Java fault caused by a call to indexOf, where a call to lastIndexOf should have been performed. Our experiments, which used a C corpus, found multiple occurrences of the analogous C transformation: a strchr $\Rightarrow$ strrchr identifier shift.

New mutation operators

- *Omit chaining method call.* Mutgen was able to identify mutation operators of this type, where the fault is a missing call to a one-argument function whose return type is equal to (or a subtype of) its argument's type. Specifically, it found patches in which a missing call to an SQL string-sanitization function was inserted.

- *Direct access of field.* While we were unable to find this mutation operator among the harvested operators—most likely because we were using only C patches—this mutation category could be generated by a combination of an identifier shift and a syntactic mutation operator.

Other mutation operators

- *Specific literal replacements.* The idiomization technique (Section 3.2) allows mutgen to identify specific literals to be used in mutation operators. To iden-

Table 3.3: Experimental results when harvesting from backward or forward patches

Aspect	Backward	Forward
Extracted syntactic mutation operators	7570	8069
Extracted identifier shifts	5000	5000
Total number of syntactic mutants	139 289	183 683
Total number of applied identifier shifts	1876	1876
Successfully compiled syntactic mutants	20 803	21 617
Successfully compiled identifier shifts	127	127
Compilation rate	15%	12%
Average Am(S)	0.81	0.81
Median Am(S)	0.81	0.81

tify literals that are more relevant to the system under test, the implementation allows the system under test to be used as its own source of idioms.

Figure 3.6 illustrates that these operators are all within the harvesting capabilities of mutgen. Just et al. provide diff-formatted patches to illustrate faults not coupled to existing mutation operators; mutgen is able to harvest mutation operators automatically from the provided patches.[6] The patches provided by Just et al. were in Java; while our experiments exclusively used C, our toolchain is language agnostic and we were able to create a Java language-definition file and extract mutation operators from the provided Java patches. In addition to being able to harvest such mutation operators from diff-formatted patches, a user of our system can also manually specify additional mutation operators in all of the above categories.

Research Question 3: Do Wild-Caught Mutation Operators Differ From Existing Mutation Operators?

Research Question 3 asks whether wild-caught mutants exhibit behavior that is quantifiably different than existing mutation operators. Table 3.3 summarizes some

[6]For some of these patches, it is necessary to supply command-line arguments to change the values of mutgen's options from their defaults—specifically, those relating to total-identifier count and the commonality threshold for harvesting identifier shifts.

```
- return solve(min, max);
+ return solve(f, min, max);
```

(a) Math-369 fix as found in Just et al. (2014b)

```
- :return $1 .( $2 ., $3 .) .;
+ :return $1 .( $_ ., $2 ., $3 .) .;
```

(b) Math-369 fix as generalized by mutgen

```
- int indexOfDot = namespace.indexOf('.');
+ int indexOfDot = namespace.lastIndexOf('.');
```

(c) Closure-747 fix as found in Just et al. (2014b)

```
indexOf ⇒ lastIndexOf
```

(d) Closure-747 fix as generalized to an identifier shift by mutgen

```
- return ... + toolTipText + ...;
+ return ... + ImageMapUtilities.htmlEscape(toolTipText) + ...;
```

(e) Chart-591 fix as found in Just et al. (2014b)

```
- :return ... .+ $1 .+ ... .;
+ :return ... .+ $_ .. $_ .( $1 .) .+ ... .;
```

(f) Chart-591 fix as generalized by mutgen

```
- FastMath.pow(2 * FastMath.PI, -dim / 2)
+ FastMath.pow(2 * FastMath.PI, -0.5 * dim )
```

(g) Math-929 fix as found in Just et al. (2014b)

```
- $1 .. :pow .( :2 .* $1 .. $3 ., .- $4 ./ :2 .)
+ $1 .. :pow .( :2 .* $1 .. $3 ., .- :0.5 .* $4 .)
```

(h) Math-929 fix as generalized by mutgen, with "pow," "2," and "0.5" keywords added by idomization

Figure 3.6: Examples of mutation operators proposed by Just et al. (2014b) and identified by mutgen

```
- return getPct((Comparable<?>) v);
+ return getCumPct((Comparable<?>) v);
```

(i) Math-337 fix as found in Just et al. (2014b)

```
getPct ⇒ getCumPct
```

(j) Math-337 fix as generalized to an identifier shift by mutgen

```
- lookupMap = new HashMap<CharSequence, CharSequence>();
+ lookupMap = new HashMap<String, CharSequence>();
```

(k) Lang-882 fix as found in Just et al. (2014b)

```
- $1 .= :new $2 .< $3 ., $3 .> .( .) .;
+ $1 .= :new $2 .< $_ ., $3 .> .( .) .;
```

(l) Lang-882 fix as generalized by mutgen

```
- if (u * v == 0)
+ if ((u == 0) || (v == 0))
```

(m) Math-238 fix as found in Just et al. (2014b)

```
- :if .( $1 .* $2 .== :0 .)
+ :if .( .( $1 .== :0) .|| .( $2 .== :0 .) .)
```

(n) Math-238 fix as generalized by mutgen, with "0" keyword added by idiomization

Figure 3.6: Examples of mutation operators proposed by Just et al. (2014b) and identified by mutgen

basic metrics from the mutation-testing experiment with Space.

Mutation-Detection Ratio Andrews et al. (2005a) defined the sample-based mutation-detection ratio Am(S) (see definition 3.1 in section 3.3), and measured it as 0.75 when existing mutation-testing techniques were applied to Space (Do et al., 2005) and Space's test suite. Using the same sample-based technique, we measured an Am(S) value of 0.81 for the mutation-operator set created via the wild-caught-mutants technique. This indicates that the mutation operators obtained by the wild-caught-mutants technique lead to mutants that roughly as hard to kill as mutants from traditional mutation operators.

Compilability Using the wild-caught mutation operators, the compilation-success rate of the mutants created for Space was around 14% (see table 3.3). Although, this rate is substantially larger than our original guess that the compilation-success rate would be less than 5%, the rate is comparatively low: Andrews et al. (2005a) reported a compilation-success rate of 92% for Space. However, because of the large number of mutation operators harvested, mutation testing via wild-caught mutants still appears feasible; our set of 34 439 compilable mutants is more than three times larger than Andrews et al.'s 11 379-mutant set (34 439 = 20 802 forward mutants + 21 617 backward mutants - 7980 duplicate mutants).

The majority of failed compilations (64%) arise from simple parsing errors. Another 21% fail because mutation has turned the left operand of an assignment into a non-assignable expression (i.e., not a C *lvalue*). Other frequent compilation errors include 5% due to invalid operands to binary operators (e.g., "+" applied to a pointer and a double) and 3% due to using an undeclared identifier. Compilation errors of these kinds are to be expected, given the lexical level at which we operate. Traditional mutation operators limit changes to ones that are unlikely to ever introduce parsing errors. For example, negating an `if` condition or replacing a "<" with a "<=" will not break compilation except under truly exceptional circumstances. Thus, the high compilation rates of traditional mutants arise essentially by construction. The wild-caught-mutants approach offers no such guarantees. That

means we waste more time on failed compilations, but it also means that we have the potential to change code in much more interesting ways.

Research Question 4: Are "Forward" and "Backward" Patches Different?

While considering patches in the backward direction ("backward patches") intuitively seems more likely to (re)introduce bugs, which is good from the standpoint of mutation testing, we also tried harvesting mutation operators by considering the same set of patches in the forward direction ("forward patches").

Overlap We found that the overlap is considerable between the sets of mutation operators harvested by considering patches in the "forward" and "backward" directions, but there is significant non-overlap: of the 13 929 unique mutation operators found using both techniques, 5860 were found only from backward patches, 6359 were found only from forward patches, and 1710 were found by both techniques.

Mutation-Detection Ratio The mutants caused by mutation operators harvested by forward and backward patches are ultimately equally difficult to kill: average and median $Am(S)$ scores are 0.81 in each direction, per table 3.3. This may seem surprising, if backward patches truly represent bug reintroduction. However, one must keep in mind that the "gold" version of Space used in our experiments passes its entire, extensive test suite. The test suite, then, effectively traps Space into a rather narrow set of allowed behaviors. Any deviation from that, whether to fix a fault or not, is likely to trigger at least one test case failure. Given the constraints of an extensive test suite, any change will look like a new fault, whether derived from backward or forward patches. Ultimately, forward patches may still describe interesting human-generated changes, and therefore harvesting them can be a worthy enhancement to backward-patch harvesting.

Ability to Reproduce Faults in Space To evaluate the differences between mutation operators harvested from forward and backward patches, we examined the faults in the 38 faulty versions of Space. We classified each fault as to whether the two kinds of harvested mutation operators could reintroduce them, if mutation testing were carried out on the "gold" version.

- Seven faulty versions (3, 4, 5, 6, 20, 21, and 28) had faults that would be reintroduced by some mutation operator harvested from backward patches.
- One faulty version (30) had a fault that would be reintroduced by a mutation operator that was harvested from both the "forward" and "backward" patches.
- Five faulty versions (1, 2, 18, 23, and 33) had faults that could potentially be reintroduced by mutins, but with mutation operators that were not harvested—in either direction—from the 50 GitHub projects that we analyzed. Of the five, faulty version 18 had a fault that could potentially be reintroduced via an identifier shift, albeit one that we did not harvest; the faults in the remaining four are expressible as syntactic mutation operators.

The remaining 25 faulty versions required mutations outside of the scope of our current techniques. The majority of these are expressible as syntactic mutation operators, but involve too many lexical tokens to survive our filtering heuristics.

While these results are limited in scope, they provide weak support for the conjecture that, compared to forward-harvested mutation operators, backward-harvested operators can introduce defects that more closely resemble defects introduced by real programmers. Ultimately, while the *ideal* is to insert "bug-like" changes into the target program, a robust test suite must also be able to identify behavior changes introduced by human programmers, which our wild-caught mutants—whether derived from bug fixes or not—simulate.

3.5 Threats to Validity

There are several threats to the validity of our work.

We employ small changes—10 lexical tokens or fewer, and typically single-line—to generate our mutation operators. These size-limited patches represent a distinct subset of all possible changes to code, and as a result we do not derive mutation operators from all valid patches observed. We enact this limitation because the more complex each individual harvested mutation operator is, the less likely it is to be matched in any particular piece of code to which it is applied. Smaller and simpler mutation operators yield a substantially higher proportion of matchable mutation operators; syntactic mutation operators larger than those we harvest clutter the system, but are rarely able to be applied to a system under test.

The idealized goal of mutation testing is to measure a test suite's quality, by measuring its ability to detect faults of the kind that might be inserted into the program in the future. One may question whether *reversals of past changes* are good candidates as predictors of the kinds of future faults that one wants the test suite to detect. Our experiment with the 38 faulty versions of Space provides a small amount of evidence that backward-patch harvesting is a better source of such candidates than forward-patch harvesting.

Even if our specific harvesting approach proves to be sub-optimal, the general idea of supporting mutation testing using information harvested from a revision-control system would still have much potential. A possible improvement, which we plan to investigate in follow-on work, is to extend the harvesting operation to include information from a bug-tracking system, such as Bugzilla (Bugzilla development team, 2016). Śliwerski et al. (2005) investigated how the combination of a revision-control system and a bug-tracking system provides a way to identify fix-inducing patches in the revision history.[7] Such an approach would provide three sources of input for harvesting mutation operators: (i) the fix-inducing patch; (ii) the corrective patch; and (iii) the commonalities between the fix-inducing patch and the corrective patch.

Rather than experiment shallowly across a large benchmark suite, we chose to focus on evaluating in depth a single application: Space, from the SIR repository (Do et al., 2005). This decision allowed us to make direct comparisons with the empirical

[7] As defined by Śliwerski et al., a *fix-inducing patch* is one that causes a later bug fix.

findings of Andrews et al. (2005a). However, if Space is unlike other real-world code, then this difference would harm the external validity of our findings—i.e., the extent to which our conclusions can be generalized to other situations. In spite of that risk, Space is an appealing subject for an experiment on the effectiveness of mutation testing. It is not a synthetic benchmark, but rather is a mature piece of software that has been subject to years of production use. Among the programs studied by Andrews et al., only Space had variants with real faults instead of hand-introduced ones. Moreover, at 9124 lines of code, Space is larger than the programs in the Siemens suite. For Space, mutins generated 241 517 single-mutation mutants, of which 34 439 were compilable. The set of 241 517 mutants is a non-trivial set, but was still small enough that, for each mutant, we could run 5000 100-case test-suite subsets.

3.6 Related Work

To the best of our knowledge, the wild-caught-mutants technique is novel; however, several other projects have used related ideas. Some of the latter techniques could be used to enhance our methods for extracting mutation operators.

Śliwerski et al. (2005) describe a technique to identify fix-inducing patches within a revision history, and propose applying similar tactics to identify failure-inducing patches. Many others have used similar strategies, all based on recognizing specific keywords (such as "fixed" or "bug") or bug IDs (such as "#42233") in commit messages (Kim and Ernst, 2007; Boogerd and Moonen, 2008; Fischer et al., 2003; Mockus and Votta, 2000; Čubranić and Murphy, 2003). Mutgen could be extended to use these techniques to attempt to identify "higher-quality" mutation operators by inferring properties of the changes induced by specific patches in the source corpus.

Le et al. (2016) mine revision histories to extract bug-repairing patches, and use these as a basis for program repair. We provide a technique that, effectively, does the opposite—we use mined patches to break code instead of fixing it.

Coccinelle's *semantic patches* are generalized patches that can be applied to code, much like our syntactic mutation technique (Padioleau et al., 2006). However, Coccinelle works with manually authored patches, whereas we harvest new mutation operators automatically. Coccinelle applies semantic changes to multiple blocks of code, and has been applied to bug detection (Lawall et al., 2010, 2009), whereas we focus on breaking code for the purpose of mutation testing. The mutation-testing context lets our toolchain utilize simpler patches, as well as harvest them automatically.

As a follow-on to Coccinelle, Palix et al. (2010) developed *Herodotos*, a system to track the evolution of patterns in code through analysis of a revision history. We share Palix et al.'s interest in the evolution of code, although we focus on pairwise diffs between adjacent revisions rather than entire revision histories. Herodotos requires more manual intervention than our toolchain, most notably to create and generalize the initial patterns to be tracked across revisions. This approach is sensible for Herodotos, which ultimately drives interactive code-understanding tools. However, our batch-testing usage scenario calls for a fully automated approach.

Nam et al. (2011) describe a technique for identifying bug-fixing commits in a source-control repository and calibrating mutation testing to utilize mutation operators that more closely resemble the reverse of changes observed in bug-fixing commits. Nam et al. look for keywords in commit messages, as many others have done, and also manually inspect commits to confirm that they are indeed fixes. Our approach is more automated, as we harvest all patches that fit our purely syntactic filtering heuristics. Likewise, Nam et al. craft several new mutation operators by hand, whereas our approach automates the entire process of harvesting and generalizing new mutation operators. Our automation-focused approach may be less selective, but it allows us to work with a corpus two orders of magnitude larger than that used by Nam et al.

3.7 Experimental Artifacts

Our core mutation tools, consisting of mutgen and mutins, are available at `https://github.com/d-bingham/wildcaughtmutants`.

We also provide tools to demonstrate our experiments at `https://github.com/d-bingham/fse2017artifact`. However, reproducing our complete set of experiments would require months of processor time (and as such was executed on a high-throughput computing platform). Therefore, this experimental artifact recreates scaled-down versions of the experiments described in section 3.3.

The artifact allows the user to harvest a set of mutation operators from scraped GitHub repositories (omitting the full Linux kernel source due to space and time concerns). Once a set of mutation operators are harvested from this corpus, the artifact then generates thirty randomly-chosen mutants (chosen as mutation operators and insertion indices into the Space program), attempts to compile them, and evaluates the mutated programs against Space's test suite. With the pass/fail results from each test case, the artifact then generates random "virtual" test suites to calculate $\mathrm{Am}(S)$ scores for the generated mutants.

The artifact can be executed via a provided shell script or through the use of a Docker(Merkel, 2014) container, allowing demonstration of a small portion of our experiment in a highly portable manner.

3.8 Conclusion

For mutation testing to provide a useful measure of the sensitivity of a test suite, it must produce not only faults within the system under test, but faults that mimic those caused by the actual developers working on a project. Just et al. demonstrated that faults introduced through mutation testing can serve as proxies for real faults introduced by developers and be effectively used to evaluate the sensitivity of a testing suite, although they also described limitations of existing sets of mutation operators. We expand upon that work by automatically harvesting mutation operators—wild-caught mutants—and comparing the capabilities of the harvested

mutation operators to those of existing mutation operators.

Andrews et al. (2005a), discussing threats to validity, caution that “It is also expected that the results of our study would vary depending on the mutation operators selected....” Our findings provide strong empirical support for this expectation. As opposed to existing “synthetic” mutation testing techniques, every mutation we create is based on a (reversed) change that some real programmer made to some real piece of code. Our wild-caught approach produces novel mutation operators, in turn creating defects that are about as difficult to kill as those arising from existing synthetic mutation operators or Space’s 38 naturally-arising faults. Whether existing synthetic mutation operators or our wild-caught mutation operators can objectively be characterized as more “realistic” remains an open question.

“Realism” arguments aside, it is clear that developers benefit if their test suites can be challenged by bugs that resemble those they might expect programmers to introduce. Our wild-caught-mutants technique can be a source of such bugs. Instead of crafting mutation operators by hand, we believe that our results demonstrate that wild-caught mutants provide a stronger method for evaluating the sensitivity of test suites.

4 GUIDED MUTATION TESTING

4.1 Introduction

High-quality test suites are one of the most valuable tools available to the modern software engineer. Unfortunately, there are substantial costs associated with the development of test suites—to say nothing about the costs associated with *improving* existing test suites. Mutation testing is a technique that allows a practitioner to both assess the quality of their test suite and improve it, but it has substantial—often prohibitive—costs associated with it.

Most mutation-testing frameworks function by randomly modifying the source code of the system under test. These frameworks consist of a set of rules, called *mutation operators*, which are applied to the source code at various points, creating a new, modified program called a *mutant*. This mutant is then tested against the project's test suite. If the test suite is able to detect the mutant (via a failed test case), the suite is said to have *killed* the mutant; this result provides evidence to the engineer arguing for the robustness of the test suite. A live mutant, however, provides an example program usable by the engineer to expand the test suite—so long as the mutant program is semantically different from the original program and an input that kills the mutant can be found to create a new test case.

This technique, however, is fundamentally limited by its computational cost. On the scale necessary for robust mutation testing, compilation itself is a substantial burden. Once a mutant is created and compiled, the test suite must be executed. Because the time needed to execute the test suite can be arbitrarily high, another computational roadblock is created. This disadvantage can manifest in the unfortunate situation that without the use of *tremendous* computational resources, the results of an execution of a mutation-testing system can be out of date before completion. That is, the process can take so long that the underlying code has changed significantly through normal development in the time required for the mutation-testing system to operate. This computational barrier significantly limits the wider adoption of mutation testing.

Our aim is to create techniques that improve the real-world applicability of mutation testing by lowering these computational barriers. We propose techniques that employ machine learning to allow mutation testing to provide useful feedback while consuming fewer computational resources. Specifically, our high-level goal is as follows:

> Improve the applicability of mutation testing through the development of techniques that prioritize mutants for testing.

We have developed techniques for training machine-learning models to predict qualities important to the usability of mutation testing. Using these predictions, instead of exhaustively executing or randomly sampling from among all possible mutants that could be created from a system under test, an engineer could instead sample from a prioritized set of mutants. Testing this prioritized subsample of all possible mutants would then allow the engineer to use mutation testing in a reasonable amount of time.

The most basic prediction provided by this technique is the prediction of whether the result of applying a particular mutation operator will successfully compile. We use as an underlying mutation-testing system Brown et al.'s *wild-caught mutants*(Brown et al., 2017), a toolchain that provides robust modeling of possible changes to code, but at the cost of a lower compilation rate than other mutation-testing systems. We wish to improve on this system by preemptively avoiding mutants expected not to pass the compilation step in the process:

> Research Question 5: Can a machine-learning model predict compilation of a mutant?

Semantic equivalence of mutants is an open problem within the field of mutation testing. While equivalence testing is undecidable, in general, approximations are possible. Specifically, mutated code is semantically equivalent to the original code if the compiled version is identical. We wish to use this approximation to reduce

the number of potential non-productive mutants:

> Research Question 6: Can a machine-learning model predict the binary in-equivalence of the compiled version of a mutant and the compiled version of the code from which the mutant was generated?

In addition to successful compilation and binary inequivalence, we aim to predict a mutant's interaction with its project's test suite. While an equivalent mutant is not interesting to the engineer, a mutant that causes the program's main function to immediately exit is similarly uninteresting. Ideally, we would like to generate mutants that are both unlikely to be equivalent *and* unlikely to be "catastrophic"—in the sense that they cause such large-scale failures that they are trivially detected by *any* reasonable test suite. To quantify a mutant's likelihood to be both inequivalent and non-trivial to kill, we define the concept of *discernment* in section 4.2. We aim, then, to identify mutants with greater discernment.

> Research Question 7: Can a machine-learning model successfully predict the discernment of a mutant?

Finally, because our technique uses machine learning to make these predictions, we investigate different scenarios for using our techniques. Our techniques allow the use of either "off-the-shelf" trained models *or* models trained directly from the system under test. Thus:

> Research Question 8: How effectively do our machine-learning models generalize across projects, and what advantage is obtained by training the models on the system under test?

The contributions of our work can be summarized as follows:

We describe a set of techniques for training machine-learning models to predict

qualities of mutants created for mutation testing. Through the use of models trained with these techniques, an engineer carrying out mutation testing can generate "interesting" mutants for testing a test suite.

We created a toolchain for prioritizing mutants. This toolchain allows the user to use either "off-the-shelf" or custom-trained models to prioritize mutants generated by a mutation-testing framework. We use Brown et al.'s wild-caught-mutants system (Brown et al., 2017) as the underlying mutator. Other mutation-testing frameworks can be adapted for use with the technique. Although most other systems do not suffer from the low compilation rate of wild-caught mutants, the prediction of binary inequivalence and discernment is more universally applicable.

We report on experiments in which we analyzed five Java-language projects with voluminous test suites. Models were trained in three different modalities using mutants derived from each project individually and as a group; these models were used to determine both how well each individual model functions on its source project, as well as how well it generalized to other projects. From these data we establish both the cost and benefit of custom-training our models to individual projects.

Organization. The remainder of this chapter is organized as follows: Section 4.2 describes the conceptual layout of both our tools and the predictions they make. Section 4.3 presents the modalities used for training our machine-learning models. Section 4.4 describes the operation of the underlying mutation-testing system, our technique for acquiring training data, and the specifics of the machine-learning model our technique uses. Section 5.6 describes our experimental setup. Section 5.7 presents our experimental results. Section 5.8 considers threats to the validity of our technique. Section 5.9 discusses related work. Section 3.7 describes the artifact we plan to submit.

4.2 Shepherds and Prediction Goals

Our ultimate goal is to improve mutation testing through prioritization. Mutation testing itself is an extremely computationally intensive task, but the required

Table 4.1: Projects Used

Project	LOC	Test cases	Mutants
apache/commons-lang	80 001	5796	844 815
apache/commons-math	180 659	4864	2 738 898
google/closure-compiler	246 676	16 684	6 199 405
jfree/jfreechart	219 183	2176	3 381 303
charite/jannovar	50 966	256	1 425 961

	Single-mutant means		All mutants	
Project	Compilation	Test-suite execution	Generation	Compilation (estimated)
apache/commons-lang	26s	2m 16s	19s	293d
apache/commons-math	1m 58s	37s	1m 2s	10yr 100d
google/closure-compiler	2m 3s	2m 56s	1m 45s	23yr 216d
jfree/jfreechart	1m	22s	32s	6yr 171d
charite/jannovar	31s	1m 57s	27s	1yr 121d

	All test suites
Project	Execution (estimated)
apache/commons-lang	3yr 225d
apache/commons-math	3yr 78d
google/closure-compiler	35yr 142d
jfree/jfreechart	2yr 131d
charite/jannovar	5yr 155d

processing time is concentrated in compilation and test-suite execution. Table 4.1 shows the typical compilation and test-suite execution times required for a single mutant from each of the projects used in our experiments, as well as the time taken to identify *all* mutants that the toolchain is capable of generating—over one million mutants for most of the projects. It is notable that for all projects used in our experiments, identifying all possible mutants took less time than compiling a *single* mutant.

For all of our projects, identifying all possible mutants takes less than one-millionth the time that it would take to compile and execute their test suites. Because we have the ability to generate mutants much more rapidly than we can test them, prioritizing mutants for testing allows an engineer to spend computational resources more effectively, by concentrating on mutants predicted to be useful. We call the process of prioritizing mutants based on predicted qualities *guided mutation testing*, and a program that performs the task a *shepherd*. Thus, our work can be summarized as building a shepherd and evaluating its effectiveness at guiding the process of mutant generation.

The remainder of this section describes the qualities of generated mutants predicted by our shepherds.

Compilation

The most basic feature predicted by our shepherds is compilation. That is, for a specific mutant generated, will it be successfully compiled by its project's build system? While prediction of compilation is not universally useful due to high compilation rates (or, in cases like AST-based mutation, 100% compilation rates by construction), it benefits the wild-caught-mutants framework that we used in our experiments.

Brown et al. (2017) reported a compilation rate of roughly 14% with the C-language Space program. Our results with Java-language projects were considerably higher, averaging 49%, but that higher rate still allows for a two-fold reduction in compilation time due to avoiding non-compiling mutants.

We believe that Brown et al.'s approach has advantages over other mutation-testing suites—chief among them being language-independence—and our technique allows the wild-caught-mutants suite to function with an *effective* compilation rate closer to other suites by avoiding mutants predicted to fail compilation.

Binary Inequivalence

Detection of semantically equivalent mutants continues to be an open problem in mutation testing. While the most-strict version of the problem is undecidable, it is possible to detect some subset of semantically equivalent mutants. We identify one particular subset—those mutants that, when compiled, produce the same bytecode as their original version. We call this subset of semantic equivalence *binary equivalence*, and investigate the ability of our toolchain to detect it.

Because we are devising shepherds for which a *positive prediction* about a mutant indicates that a mutation-testing tool should *continue to work* with the mutant, henceforth we will refer to such a shepherd as predicting *binary **in**equivalence*.

To compare generated bytecode, we extract Java class files from Java jar files generated through the build process. While jar files contain timestamps and other metadata not directly derived from the underlying Java source code, the class files extracted from the jar files contain no "noise"—every byte in the file is either a strictly defined file header or bytecode directly compiled from source code.

Discernment

All generated mutants—even those compilable and identified as binary-inequivalent to their parents—are not equally useful. Petrovic et al. (2018) discuss industrial use of mutation testing in detail and define the concept of a *productive* mutant. They define a killable—that is, inequivalent—mutant to be productive if it allows the creation of an effective test case.[1] We expand upon the idea of classifying generated mutants by their fundamental usefulness with the concept of *discernment*.

[1]They also consider some equivalent mutants to be productive, but a discussion of that subset of mutants is outside the scope of this chapter.

We use discernment to quantify how "targeted" a mutation is. That is, a more-discerning mutant triggers *fewer* test cases than a less-discerning mutant. Consider the case of what we call a "thorough" test suite—one large enough for which a typical live mutant is actually equivalent to the program from which the mutant was generated. We assume that our shepherds are trained on projects with thorough test suites, and that it is reasonable to treat live mutants as semantically equivalent. We then focus on mutants that are *harder to kill* as opposed to those that are *live* by the test suite.

By predicting discernment, we can spend valuable processing time on mutants we expect to be more difficult to kill. As a concrete example, early on in the experimental process we identified mutants that disabled core output routines, and as such caused *all* test cases to fail. While these mutants consumed the same amount of processing time for compilation and test-suite execution as any others, they are *unproductive* in the sense of Petrovic et al.'s work—they neither demonstrate any particularly useful characteristic of the project's test suite nor allow for the construction of new and useful test cases.

For some test suite T containing $|T|$ test cases, we define discernment for a mutant that causes failures in x test cases as follows:

$$\mathrm{disc}_{|T|}(x) = \begin{cases} 0 & x = 0 \\ 1 - \log_{|T|} x & x > 0 \end{cases}$$

In effect, a mutant that triggers exactly *one* test case has a discernment of 1, and the value drops off logarithmically to 0 if the mutant triggers *all* test cases. Figure 4.1 plots discernment curves for $|T| = 100$, 1000, and 5000. Note that by the definition of our discernment function, all of these functions exist at $(0, 0)$.

Ultimately the choice of the value at $x = 0$ is arbitrary. Our choice of $\mathrm{disc}_{|T|}(0) = 0$ is primarily influenced by what we anticipate to be our most common use case. We expect that the typical use case of our techniques will be to use shepherds *trained* on projects with voluminous test suites, but *executed* as an "off-the-shelf" solution applied to projects with less-robust test suites. Due to the robustness of

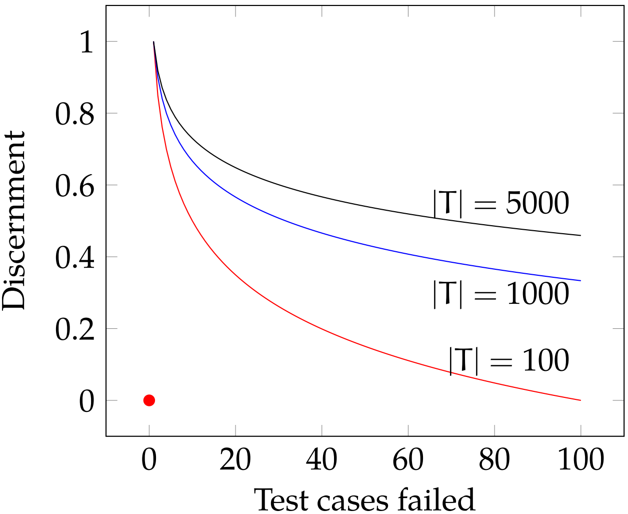


Figure 4.1: Discernment

the test suites used in our experimental corpus, even with the ability to predict binary inequivalence with reasonable accuracy, we expect that a substantial portion of mutants that do *not* cause test-case failures are semantically equivalent to their parent. As such, we choose to discourage selection of mutants triggering *zero* test cases. However, we do not expect this use case to be universal. Because the tools may be used on target projects with similar (or even greater) test-suite robustness than the training corpus, we leave the ability to change the value of $\text{disc}_{|T|}(0)$ as an option in our training-set-generation tools.

Our tools generate models that can predict the discernment of a mutant and filter mutants based on any chosen discernment threshold. Because we train shepherds to identify mutants above certain discernment thresholds, any discontinuities in the discernment function do not impede training.

Generalization and Customization

The sheer computational cost of mutation testing continues to be an obstacle to its wider use. Our work ultimately aims to improve the use of the technique by improving its efficiency, allowing an engineer to accomplish more with fewer

resources expended. By its very design, however, the use of our technique may seem to contradict these goals. We use millions of hours of computing time to train the machine-learning models on which our shepherds are built—resources unavailable to the typical software engineer. Because of this limitation, we investigate our technique's generalization and customization.

Beyond our toolchain itself, the primary artifact of our research is a shepherd trained from our experimental corpus. We analyze the capabilities of this shepherd in detail in section 5.7. Notably, we evaluate this shepherd's ability to successfully predict features of projects external to its training—in effect, treating it as an "off-the-shelf" solution to improve mutation testing *without* adding additional computation time to the project. An engineer can take this shepherd and apply it directly to other projects, as long as they use the same programming language.

However this "off-the-shelf" tool is just a starting point. Our toolchain allows for a user to train a new model from their own project, or to enhance the training set for an existing shepherd with data from their project. We investigate the costs and benefits of this customization in section 5.7.

4.3 Training Modalities

We used the wild-caught-mutants mutation-testing framework both to mutate the projects we analyzed as well as generate training-set data with which to train our shepherds. We generated multiple sets of training data at two different levels of abstraction during the experiment: mutant/concrete and mutant/abstract, described below.

Figure 4.2 shows example training data harvested during our experiment.

Mutant/Concrete

Our initial implementation of training-set generation aims to produce shepherds highly customized to the project being analyzed. We call this first training modality *mutant/concrete*. To generate these training vectors, we tokenized the Java-language

```
. toString ( ) ) ; } } builder . append ( "\"" ) ; if ( n .
```

(a) Mutant/concrete training vector (subset)

```
.. $_ .( .) .) $_ .} .} $_ .. $_ .( =_ .) $_ _if .( $_ ..
```

(b) Mutant/abstract training vector (subset)

Figure 4.2: Training modalities.

files mutated by the wild-caught-mutants framework, modifying the toolchain to produce a vector of tokens as they are processed, but before abstraction. An example of a training instance from the mutant/concrete modality is shown in Fig. 4.2(a).

Because this modality of training-set generation performs no abstraction of identifiers used in the source code, we expect shepherds trained through this method to be more accurate but less generalizable to other projects.

Mutant/Abstract

Requiring time-intensive per-project training sessions for *all* users of our technique would counter the goal of reducing the computational costs of mutation testing. To create shepherds that can more easily generalize across projects, we implemented a training modality that abstracts out identifiers and literals from the modified source code, our *mutant/abstract* training modality.

To implement this training modality, we used the built-in abstraction system in the wild-caught-mutants toolchain. The mutation operators harvested by the wild-caught-mutants toolchain are stored as a series of terms showing operators, keywords, and wildcard-style terms for iterators and literals. We utilized the abstracted tokens generated by this system to build training vectors. Fig. 4.2(b) shows an example mutant/abstract training vector; in particular, the vector shown is the abstracted version of the vector shown in Fig. 4.2(a).

4.4 Implementation

Mutation Testing

We also use the wild-caught-mutants framework as our underlying mutation-testing system. This framework "harvests" new mutation operators by mining source-code-control repository histories, focusing on commits with small changes, and generating generalized patches that represent the identified changes. These generalized patches are then used to construct new mutation operators. Each mutation operator is implemented as a "before" and "after" pattern: the insertion tool matches the "before" pattern to existing code, and then modifies the matched code to match the "after" pattern, re-using matched tokens as needed. Figure 4.3 demonstrates this process; Fig. 4.3(a) shows a small revision in a patch from a hypothetical source-control revision history, Fig. 4.3(b) shows the generalized patch abstracted from the revision, and Fig. 4.3(c) shows an example of code identified and mutated by the operator.

We chose this mutation-testing framework for three reasons. First, the framework is language-agnostic. The original instantiation of the framework for the experiments conducted by Brown et al. mutated C-language code, but we were able to adapt the toolchain to work on the Java language easily. Second, the most obvious weakness of the wild-caught-mutants technique is its low compilation rate. This limitation is largely a result of the system's language-agnostic design. Our work aims to directly improve the toolchain by offering prediction of compilation success without executing the full compilation process—overcoming one of its primary limitations. Finally, the abstracted patches allowed us to generate training data for our machine-learning model.

Parallelized Training-Set Generation

The wild-caught-mutants mutation-testing framework found over fourteen million mutants in the five projects we chose to analyze. (See table 4.1 for the numbers of mutants created for each project.) An exhaustive analysis of these mutants is beyond

```
- if (x && y)
+ if (x)
```

(a) Source patch

```
- :if .( $1 .&& $_ .)
+ :if .( $1 .)
```

(b) Extracted mutation operator

```
- if (death && taxes)
+ if (death)
```

(c) Extracted mutation operator

Figure 4.3: Wild-caught mutants.

the ability of modern desktop hardware, and even stretched the limits of the cloud-computing platform that we used in our experiments. As a result, we analyzed a subset of the entire mutation space, sampled randomly. We generated 136,858 mutants from each of the five projects, for a total of 684,290 mutants, representing about 4.7% of the mutants produced by the wild-caught-mutants framework.

Machine-Learning Model

We used long short-term memory (LSTM) models (Hochreiter and Schmidhuber, 1997) to drive our shepherds. These models were implemented with the Keras machine-learning API (Chollet et al., 2015). Each trained shepherd predicts one quality of mutants: compilation, binary inequivalence, or discernment, as described in section 4.2. Each shepherd is trained with data generated by one of the training modalities described in section 4.3. Because we train shepherds from training-set data generated from different modalities *and* to predict different qualities, we can train several different shepherds from a single source corpus. Section 5.7 presents results on how the different shepherds perform.

Training Vectors

To train our machine-learning models, we generated training vectors with the aid of the wild-caught-mutants framework. For each iteration of our training-set generation process, we randomly created a mutant from each of our chosen projects (described in section 5.6). We then attempted to compile this mutant. If it successfully compiled, we then tested it for binary inequivalence (as described in section 4.2), executed the test suite, and recorded the pass/fail status of all test cases. We then used a modified version of one of the wild-caught-mutants tools to extract a stream of tokens representing the mutant. This process was executed in parallel on a cloud-computing platform to allow us to generate sufficient data to train our shepherds.

This process generated several training vectors:

- For *each* of the 684,290 mutants used for training, we generated a training vector with the mutant's compilation status as a label, and a representation of the mutant for each of three training modalities as described in section 4.3.
- For every *successfully compiled* mutant, we generated:
  - A training vector labeled with its binary inequivalence to its parent for each of the three training modalities.
  - A training vector labeled with its calculated discernment value (as described in section 4.2) for each of the three training modalities.

Our shepherds were then trained from the training sets composed of these training vectors.

4.5 Experimental Setup and Training

Criteria used to Select Projects for Study

Because one of our goals was to evaluate how well our technique generalizes across projects, we needed to choose projects that were implemented in the same

Table 4.2: Training Results

	Correct Compilation Prediction (%)						
		mutant/abstract			mutant/concrete		
Project	Split	Ind.	Exc.	Held Out	Ind.	Exc.	Held Out
apache/commons-lang	51	90	86	74	90	86	64
apache/commons-math	54	82	81	71	90	84	70
google/closure-compiler	68	85	83	72	80	82	70
jfree/jfreechart	56	64	85	79	91	80	79
Average		*80*	*84*	*74*	*88*	*71*	*71*
	Correct Binary-Inequivalence Prediction (%)						
		mutant/abstract			mutant/concrete		
Project	Split	Ind.	Exc.	Held Out	Ind.	Exc.	Held Out
apache/commons-lang	73	84	76	71	88	88	37
apache/commons-math	64	72	68	66	76	78	58
google/closure-compiler	53	65	84	65	71	79	39
jfree/jfreechart	69	89	83	69	87	79	36
Average		*78*	*78*	*68*	*81*	*81*	*43*
	Correct High-Discernment Prediction (%)						
		mutant/abstract			mutant/concrete		
Project	Split	Ind.	Exc.	Held Out	Ind.	Exc.	Held Out
apache/commons-lang	78	80	80	72	92	76	56
apache/commons-math	77	76	64	65	79	73	57
google/closure-compiler	72	78	78	84	87	76	48
jfree/jfreechart	74	74	77	58	86	78	66
Average		*77*	*75*	*70*	*86*	*76*	*57*

language. We chose to concentrate on Java projects because of (i) the large number of open-source Java projects available, (ii) the availability of existing mutation-testing tooling for Java, and (iii) the prevalence of a common build system (Maven) within the Java community.

We sought open-source projects that met the following criteria:

- *Large-enough size.* We wanted projects large enough to be representative models of a software-engineering effort large enough to warrant the application of mutation testing. We used $\geqslant 50,000$ lines of code as the threshold on size.
- *Ease of integration.* We wanted projects that could be built easily on the cloud-computing system that we used for generating training data—without requiring large amounts of *per-project* engineering work. Projects were also required to have test suites that could be run entirely automatically.
- *Robustness of test suite.* We wanted projects that had substantial test suites, so that we would be able to calculate our discernment metric more accurately.

We found that four of the six projects that are currently part of Defects4J Just et al. (2014a) met our criteria. The first four columns of Table 4.1 provide information about these projects: GitHub project name, lines of code (LOC), test cases, and the number of mutants the wild-caught-mutants framework was able to generate from the project's source code.

Training Projects Used

We worked with four of the projects from Defects4J. Our experiments (and the artifact that we plan to submit) use the current revision of each project as of July 14, 2019.

A brief description of each project follows.

apache/commons-math

Described as a "library of lightweight, self-contained mathematics and statistics components," the Apache Commons Math library is an open-source Java-language project hosted on GitHub and maintained by the Apache Software Foundation.

apache/commons-lang

Also maintained by the Apache Software Foundation, the Apache Commons Lang library provides utility classes to supplement Java's java.lang API.

google/closure-compiler

Closure is Google's open-source Javascript optimizer. Hosted on GitHub, Closure is a tool that takes Javascript code as input and optimizes it for speed, while minimizing the size of the resulting code to reduce the code's download time.

jfree/jfreechart

JFreeChart is an open-source Java library for generating charts.

Unused Training Projects

Two of the Defects4J projects were rejected for use in our experimentation:

- *Mockito.* This project was rejected due to its use of the Gradle build system; all of the selected projects utilize both the Maven build system and the Surefire unit-test plugin. For simplicity, our experimental framework relies on both Maven and Surefire.

- *Joda-Time.* Because this project is no longer under active development, its dependencies were out of sync with the rest of the analyzed projects. We rejected this project to expedite development of the tools for generating our training-set data.

Training Process

During the creation of our shepherds, we extracted training-set data from the four projects selected from the Defects4J corpus (based on 136,858 mutants randomly selected from each project). Using this training set, we trained a collection of LSTM models. To increase the accuracy of our trained models, we tuned the settings of both our infrastructure for generating training-set data *and* the hyperparameters of our LSTM model.

Table 4.2 summarizes the results of the training process, showing the resulting accuracy of the models trained. The columns in this table represent:

- *Split*—The fraction, as a percentage, of the data set represented by the most common label (for compilation, predicting success/failure; for binary inequivalence, equivalent/inequivalent; for discernment, high/low discernment).
- *Individual*—The accuracy of a model trained on *only* data from the particular project, and evaluated on a test set sampled from the same project.
- *Excluded*—The accuracy of a model trained on all projects *except* the particular project, and evaluated on a test set sampled from those projects.
- *Held Out*—The accuracy of a model trained from the data discussed in *Excluded*, above, but evaluated on the excluded project.

The "Individual" results of the table represent use of the technique in its most customized state. That is, all machine-learning models underpinning the evaluated shepherd were trained on—and trained only on—training-set data generated from the project being evaluated.

The "Held Out" results are a proxy for real-world "off-the-shelf" use—that is, they show the accuracy of the model when predicting qualities of mutants generated from projects that the model was *not* trained on.

Once we were satisfied with the performance of the models we trained, we combined them into a final shepherd and evaluated its ability to predict qualities of

a newly selected project (see section 4.5). The results of this evaluation are detailed in section 5.7.

Evaluation Project

Because during the training process we were still tuning our models—both in terms of settings for generating training-set data and hyperparameters of the LSTM model—we chose to evaluate the model on a fifth project that we did not use for testing (or analyze in any form) until after we considered our technique and LSTM hyperparameters to be in a "final" state.

To find a suitable evaluation project, we used the BugSwarm (Tomassi et al., 2019) database, which is a second corpus of projects made available for software-engineering research. From BugSwam, we chose Jannovar (Robinson et al., 2017), an open-source Java library for annotating gene-sequencing data. While our training projects were selected in large part due to their robust test suites and relatively large size, Jannovar was selected as a more typical example of an open-source project with a smaller test suite, while still using the same Maven and Surefire tools on which our experimental framework was built.

When evaluating our technique on Jannovar, no tweaking of the system—neither for training-set generation nor hyperparameters of the machine-learning models—was performed. We chose to avoid any additional layers of tuning to better represent the use of our tools as a pure "off-the-shelf" solution. This step was done to evaluate more accurately the accuracy of our shepherds when used as a true "off-the-shelf" solution.

4.6 Results

After training our shepherds to predict the qualities described in section 4.2, we evaluated their performance for making predictions of the same qualities on the Jannovar project (section 4.5). This experiment was performed in "Held Out" style—i.e., the shepherds were trained on all four projects from Defects4J, then tested on

Table 4.3: Experimental Results on Jannovar ("Held Out")

Quality	Split	mutant/abstract (%)				
		Accuracy	TP	TN	FP	FN
Compilation	65	76	62	14	20	4
Binary Inequivalence	51	75	45	30	19	6
Discernment	73	77	20	57	7	16
		mutant/concrete (%)				
Quality	Split	Accuracy	TP	TN	FP	FN
Compilation	65	67	38	29	6	27
Binary Inequivalence	51	62	39	23	25	12
Discernment	73	25	9	16	57	18

Jannovar. Table 4.3 shows the results of this experiment, showing for each quality the ratio of positive and negative cases in the experimental set (Split), accuracy of the model, true positive rate (TP), true negative rate (TN), false positive rate (FP), ad false negative rate (FN). We find that accuracies achieved on Jannovar are comparable to, and mostly slightly better than, the average accuracies for the "Held Out" experiments reported in table 4.2, particularly for the mutant/abstract modality.

One caveat about these results must be acknowledged: we neglected to control for the size of the training-set data in this experiment. That is, the "Held Out" experiments in Table 4.2 used shepherds trained on a total of $3 \times 136,858 = 410,574$ mutants, whereas the Jannovar experiments used shepherds trained on a total of $4 \times 136,858 = 684,290$ mutants.

Research Question 1: Can Mutant Compilation Be Predicted by a Shepherd?

We achieved 76% accuracy using our shepherd to predict the successful compilation of Jannovar mutants using the mutant/abstract training modality. The shepherd

trained with the mutant/concrete training modality correctly predicted 67% of compilation results. Note that this use of mutant/concrete was specifically *not* trained on the Jannovar project; a decrease in effectiveness going from mutant/abstract to mutant/concrete is expected.

Research Question 2: Can Binary Inequivalence Be Predicted by a Shepherd?

For binary inequivalence, we observed similar results to compilation. Our mutant/abstract shepherd achieved 75% accuracy and our mutant/concrete shepherd achieved 62% accuracy.

Research Question 3: Can Mutant Discernment Be Predicted by a Shepherd?

The discernment-prediction shepherd obtained similar results to the compilation and binary-inequivalence shepherds when trained with the mutant/abstract modality, achieving 77% accuracy. However, the mutant/concrete version of this shepherd achieved abysmal results, only successfully predicting 25% of evaluated mutants.

Research Question 4: How Effectively Do Our Shepherds Generalize Across Projects?

Table 4.3 shows the ultimate result of our experiment, namely the results of predicting qualities of mutants generated from a project with a shepherd trained independently from the analyzed project. This result shows that our shepherds are able to generalize across projects. As expected, the more general mutant/abstract training modality produced shepherds more capable of generalizing across projects than the mutant/concrete modality. While all of our shepherds exhibited this gap in performance when evaluated on projects other than those they were trained on, the difference was most stark with the discernment shepherd. These results

confirm that while mutant/concrete grants the strongest performance when trained on the project being analyzed (see the "Individual" columns in table 4.2), any "off-the-shelf" use of our shepherds should be exclusively with those trained with the mutant/abstract modality.

Discussion

The broader implications of these results are that they offer a way to reduce the resources needed for mutation testing. As shown in the "All mutants: Generation" column of table 4.1, mutation operators can be applied quickly—so quickly that the time for creating mutants can be ignored.[2] Our shepherds can be used to cull mutants when the respective shepherd predicts any of the following conditions: (i) the mutant will not compile, (ii) after compilation the mutant will be binary equivalent to the compiled code of the original program, or (iii) the mutant has "low discernment," which means that the mutant is predicted to be easy to kill.

For such culling, the most readily usable result of our work is a suite of "off-the-shelf" shepherds that, while *trained* using hundreds of thousands of hours of processor time, can be installed on a typical desktop setup and executed in real time; no further training is required.

Our toolset also supports the use-case where the shepherd is custom-trained by the user. The earlier subsections of Section 5.7 describe the differences in predictive power that we found among shepherds trained with different modalities (mutant/concrete and mutant/abstract). The general strength of the mutant/abstract training modality does not, however, negate the usefulness of the mutant/concrete modality. When training a shepherd on the project itself—the "Individual" column in table 4.2—the mutant/concrete-trained shepherds outperform mutant/abstract-

[2]For AST-based mutation, one has to invoke the front-end of a compiler, so applying an AST-based mutation operator takes some fraction of full compilation time. Our work uses the wild-caught-mutants framework (Brown et al., 2017), for which the application of a mutation operator involves only text-substitution operations. As seen from the "Single-mutant means: Compilation" and "All mutants: Generation" columns of table 4.1, the time to create *all* mutants for each project (i.e., between 844K and 6.3M mutants) is comparable to the average compilation time for *one* mutant—and hence is an insignificant portion of the overall time used during mutation testing.

trained shepherds. The substantial drop-off in performance between the "Individual" and "Held Out" trials of the shepherds indicates, however, that the mutant/concrete training modality is primarily useful for shepherds trained on the projects on which they are intended to be used. Our toolset gives users the choice whether to use "off-the-shelf" shepherds or train customized, higher-performing, shepherds.

In general, a user will have to decide for themselves whether it is warranted to expend the large amount of resources required to train a custom shepherd.

Of course, all three kinds of shepherds have false positives and false negatives (see table 4.3), with the following consequences:

False positive by the compilation-prediction shepherd: some compilations of mutants will be attempted that result in a compilation failure.

False negative by the compilation-prediction shepherd: for some mutants that actually do compile, compilation will not be attempted.

False positive by the binary-inequivalence shepherd: some compilations of mutants will be attempted that produce code that is binary equivalent to the compiled code of the original program.

False negative by the binary-inequivalence shepherd: for some mutants that would result in code not binary equivalent to the compiled code of the original program, compilation will not be attempted.

False positive by the discernment shepherd: some compilations of mutants will be attempted for mutants that are easy to kill.

False negative by the discernment shepherd: for some mutants that are actually hard to kill, compilation will not be attempted.

Figure 4.4 shows an example use of our shepherds to filter the mutants generated by the wild-caught-mutants framework on the Jannovar project. In simple terms, filtered mutants mostly represent potential wasted effort (true negatives) that our

Mutants: 1 425 961
Filtered by compilation-prediction shepherd: 1 169 288 (82%)
Filtered by binary-inequivalence shepherd: 92 402 (36%)
Filtered by discernment shepherd: 119 917 (73%)
Compilation failure: 7540 (17%)
Binary equivalent: 13 253 (36%)
High-quality mutants: 23 561

Figure 4.4: Estimated Jannovar mutants filtered by shepherds, discarded by compilation, and retained at each filtering stage

shepherds let us avoid, albeit while also discarding some number of false negatives. Discarded mutants are actual wasted effort (false positives), while retained mutants are true positives: the desired, high-quality result. Overall, our trio of shepherds filter out 96.9% of mutants as likely wasted effort. Actual wasted effort accounts for just 1.4% of the initial mutant pool, with the remaining 1.7% constituting the sought-after high-discernment mutants that compile to inequivalent binaries.

As shown in table 4.1, the wild-caught-mutants framework can generate over 1.4 million mutants for Jannovar. Compiling and testing each of these mutants would take over six years on current desktop hardware, placing exhaustive mutation testing beyond the reach of virtually all developers. However, our technique allows the user to filter this million-plus set of mutants down to thousands—a set that could be processed in roughly a month on a modern desktop computer, or much faster parallelized across a computational cloud. These "high-quality" mutants—those expected to compile, exhibit binary inequivalence, and have high discernment—can then be further reduced by random sampling to produce a set of mutants of a size

reasonable to, and chosen by, the end user.

While we do observe both false positives and false negatives in the predictions of our shepherds, we believe that both of these problems are manageable. Ultimately, the set of mutants surviving the filtering process will typically still be large enough that random sampling will be used to generate a manageable set of mutants to test. Because we still have a large number of mutants after filtering, the impact of false negatives—that is, mutants excluded by the shepherds that would have been productive—is reduced. We do see false positives from the predictions of our shepherds, but save much more computational time from the true positives than is wasted through compilation and testing of mutants resulting from these false positives.

4.7 Threats to Validity

Mutation testing is *tremendously* expensive in terms of computational costs. Our approach, rather than being particularly lightweight, is so computationally expensive that we were unable to execute our experiment without the use of cloud-computing resources. At first glance, this result may seem to contradict our aims. However, our goal has been to support multiple use-cases, through both off-the-shelf shepherds and custom-trained shepherds. In particular, the "off-the-shelf, mutant/abstract" shepherds are an economical way to take advantage of our work: they were *trained* using hundreds of thousands of hours of processor time, but execute in real time on typical desktop hardware, and, as discussed in section 4.6, achieve 76%, 75%, and 77% accuracy for compilation prediction, binary-inequivalence prediction, and discernment prediction, respectively (Table 4.3).

Construct Validity A consistent danger in the evaluation of machine-learning results is that high accuracy is not necessarily indicative of a good model, especially if the testing sets used are unbalanced. Highly unbalanced training sets can lead to models exhibiting high accuracy, *e.g.*, by always reporting the most-common classification. While we were fortunate enough to have training sets relatively

equally balanced, we were cognizant of this potential pitfall. Table 4.3 details our results at a higher granularity than just accuracy itself, showing that our models in all cases are capable of identifying both true positives and true negatives.

Internal and External Validity It is difficult to describe a "typical" software-engineering project, and thus the threat is that our results may not apply to an *actual* "typical" project.

Section 4.5 described the criteria that we used to choose our training corpus. These criteria led us to projects with especially robust test suites, which may be atypical for software in the wild (i.e., this selection bias is an internal-validity threat).

To mitigate this concern, we chose our evaluation project, Jannovar, using looser criteria; in particular, we chose Jannovar primarily because it shared the same build system and testing plugin as the projects in our training corpus. Jannovar is somewhat smaller size, and has a test suite that is about an order of magnitude smaller than the others. It is intended to be more representative of a "typical" project, and hence represents a step to control for both the internal-validity threat mentioned above, as well as the external-validity threat of whether our results hold on projects outside the training corpus. Our results show that most of our shepherds have good "Held-Out" performance on our training corpus (the four projects from Defects4J). Moreover, our off-the-shelf shepherds, trained on the four projects from Defects4J exhibit good performance on Jannovar: the accuracies achieved on Jannovar are slightly better than the average accuracies for the "Held Out" experiments reported in Table 4.2.

Detection of equivalent mutants continues to be an open question in mutation testing, and we provide no silver-bullet solution to this problem. To aid the end user, we do provide the ability to filter out a subset of equivalent mutants through our shepherds' ability to predict binary inequivalence. Our discernment concept also touches on this problem, in that a user can specify during the training process whether to consider a mutant failing no test cases to be desirable or undesirable.

4.8 Related Work

Mutation Testing

Mutation testing is an active field of research, and our concern about its computational cost is not unique (Petrovic et al., 2018). Gopinath et al. (2016) discuss increases in efficiency—and theoretical limits on the maximum increases in that efficiency—from selective mutation testing, but focus their analysis on mutation-*operator* selection, while we focus on the mutation itself. Kurtz et al. (2016) describe a process for reducing the generation of undesirable mutants, but like Gopinath et al. focus their analysis on mutation operators.

Saleh and Nagi (2015) approach the problem of computational costs in a different manner entirely—by developing a mutation-testing framework built on a cloud-computing platform. While our work *does* make use of cloud-computing resources in the training phase, our primary goal is to reduce the computational cost of effective mutation testing as opposed to streamlining its use in a cloud-computing environment.

Analysis of Code With Machine Learning

Our approach centers around leveraging established machine-learning techniques to deepen understanding of code. This concept is not alien to the wider programming-languages community. Hellendoorn et al. (2019) apply a recurrent neural network (RNN)—our model, an LSTM, is a form a RNN—to propose invariants to be used in the analysis of programs. Hellendoorn and Devanbu (2017) train machine-learning models (including LSTMs) with tokenized source code in much the same way as our shepherds are trained, however their goal is the prediction of subsequent tokens; we, instead, train our models to predict global attributes of the analyzed mutant.

4.9 Conclusion

Exhaustive approaches to mutation testing continue to be too computationally intensive to see wide adoption (Petrovic et al., 2018), with random sampling of mutants being one of the most effective methods used to reduce this computational cost (Gopinath et al., 2016). If mutation testing is to be adopted by the industry at large, it will need to conform to *reasonable* limits on its computational cost. That said, as the needs—and resources—of software engineers across the industry vary wildly, so does the definition of *reasonable*.

In this chapter we describe a technique for improving the utility of mutation testing through the prioritization of mutants. Because the *generation* of mutants requires vastly fewer resources than their evaluation, we provide a set of tools—shepherds, as we call them—for rapid analysis of these generated mutants to filter down the massive set of potential mutants to a much smaller set of "high-quality" mutants. Depending on the resources available to the engineer using our technique, these filtered mutants can then be further reduced by random sampling. We provide both an "off-the-shelf" version of our shepherds, as well as the tools to custom-train shepherds for specific projects. Custom-trained shepherds are likely to provide greater prediction accuracy at the cost of more resources spent in training. By providing both pre-trained shepherds as well as tools for custom-training shepherds, we address the problem of mutation testing's high computational cost by providing the engineer with a series of options that can be tailored to their available resources.

5 TEST-CASE GENERATION

5.1 Introduction

Our aim is to provide a tool-chain for test-case generation that generates not only *new* but *effective* test cases. We propose a technique that employs mutation testing to identify weaknesses in existing test suites, and then applies target-oriented fuzzing to generate test cases that plug these holes, resulting in a more robust test suite—that is, one that is (typically) larger and more effective at detecting faults. Our tool-chain combines these techniques to provide an engineer not only with new test cases but also with evidence for their utility—specifically, for each generated test case, a live mutant (with relation to the existing program and existing test suite) and a new input to the program that kills the mutant. (Mutants identified as having different behavior from the original program are said to be *killed*, while those undetected by the test suite are *live*.) Our overall hypothesis is as follows:

> The combination of mutation and target-oriented fuzzing can be used to not only generate test cases, but generate effective test cases.

We assume that a program P accepts, as its input, some data C^{in} and, upon execution, returns some output C^{out}. Assuming this input is a series of bytes, there is an obvious and trivial algorithm for generating new test cases: randomly select a sequence of bytes C^{random}, record the program's output: $C^{out_random} = P[C^{random}]$; and call the tuple $\langle C^{random}, C^{out_random} \rangle$ thus created a test case.

Unfortunately, unless one is *only* interested in testing the input validation of P, this test case is unlikely to be useful. Additionally, as software-engineering projects mature and their test suites grow in size, the resources consumed by their execution increasingly becomes a liability. This limitation—the raw amount of processing power and/or time required for the execution of test suites—further restricts the real-world value of a test case generated primarily through random means.

With only finite resources available for test-suite execution, it is desirable to have test cases that are *useful* to the project. Our work focuses on not only the generation of useful test cases, but the ability of our tool-chain to present an engineer maintaining a project an *argument* justifying the usefulness of the generated test cases. Our tool-chain uses mutation-testing to identify holes in existing test suites, and attempts to plug them with inputs generated through target-oriented fuzzing.

In pursuit of this goal (and to evaluate the tool-chain we have developed), we address the following research questions:

First, to generate test cases (and the arguments to the engineer supervising their potential admission into a test suite), we must be able to mutate the program successfully, identify live mutants, and then kill them through target-oriented fuzzing. Once mutants are created, we evaluate the ability of our tool-chain to generate effective test cases based on these mutants.

> RESEARCH QUESTION 9: Can target-oriented fuzzing be used to create test cases to kill live mutants?

Next, we evaluate the ability of our tool-chain to generate not only individual test cases but entire test suites.

> RESEARCH QUESTION 10: Can target-oriented fuzzing combined with mutation analysis be used to generate robust test suites?

Finally, we use our tools to automatically generate test cases in bulk, and compare the resulting test suite with human-authored test suites.

> RESEARCH QUESTION 11: How do these generated test suites compare to manually-authored test suites in size, code coverage, and the ability to kill mutants?

The contributions of our work can be summarized as follows:

We describe a technique for creating new and *useful* test cases for a program. We identify potential new test cases via mutation-testing of the program, using target-oriented fuzzing to create a new test case for each live mutant.

We created a tool-chain for generating these test cases. Our tool-chain allows an engineer to "plug-in" a mutation-testing system to create live mutants, and then generate new test cases by killing these mutants.

We report on experiments in which we generated new test cases for the SPACE program, and constructed several test suites for it. We used this process to determine how well our technique can both generate an individual test case (given a live mutant) and build an entire test suite from a minimal starting point.

Our tool-chain is able to construct both individual test cases, as well as entire test suites. Moreover, it can construct a test suite that has similar performance characteristics to a hand-constructed test suite—as measured by size, code coverage, and the ability to kill mutants—but at a fraction of the cost in time and dollars.

Organization. The remainder of the chapter is organized as follows: Section 5.2 motivates the research. Section 5.3 states the problem that we address. Section 5.4 describes our algorithms. Section 5.5 provides an overview of target-oriented fuzzing. Section 5.6 describes our experimental setup. Section 5.7 presents experimental results. Section 5.8 considers threats to validity. Section 5.9 discusses related work. Section 3.7 describes the artifact we plan to submit.

5.2 Motivation

Test-suite development often focuses on code coverage, but this strategy has limits (Inozemtseva and Holmes, 2014). Instead, we attempt to construct stronger test suites through the combination of mutation testing and target-oriented fuzzing.

Mutation testing is a well-known technique used to evaluate the quality of a test suite (Jia and Harman, 2010). For a given program and its associated test suite, mutation testing determines the *mutation-adequacy score* of a test suite by creating a set of mutants (variants of the program under test, constructed by making small changes to the original program) and determining which of these mutants the test

suite can correctly identify as having behavior different from the original program. The higher the mutation-adequacy score, the greater the ability of the test suite to kill mutants.

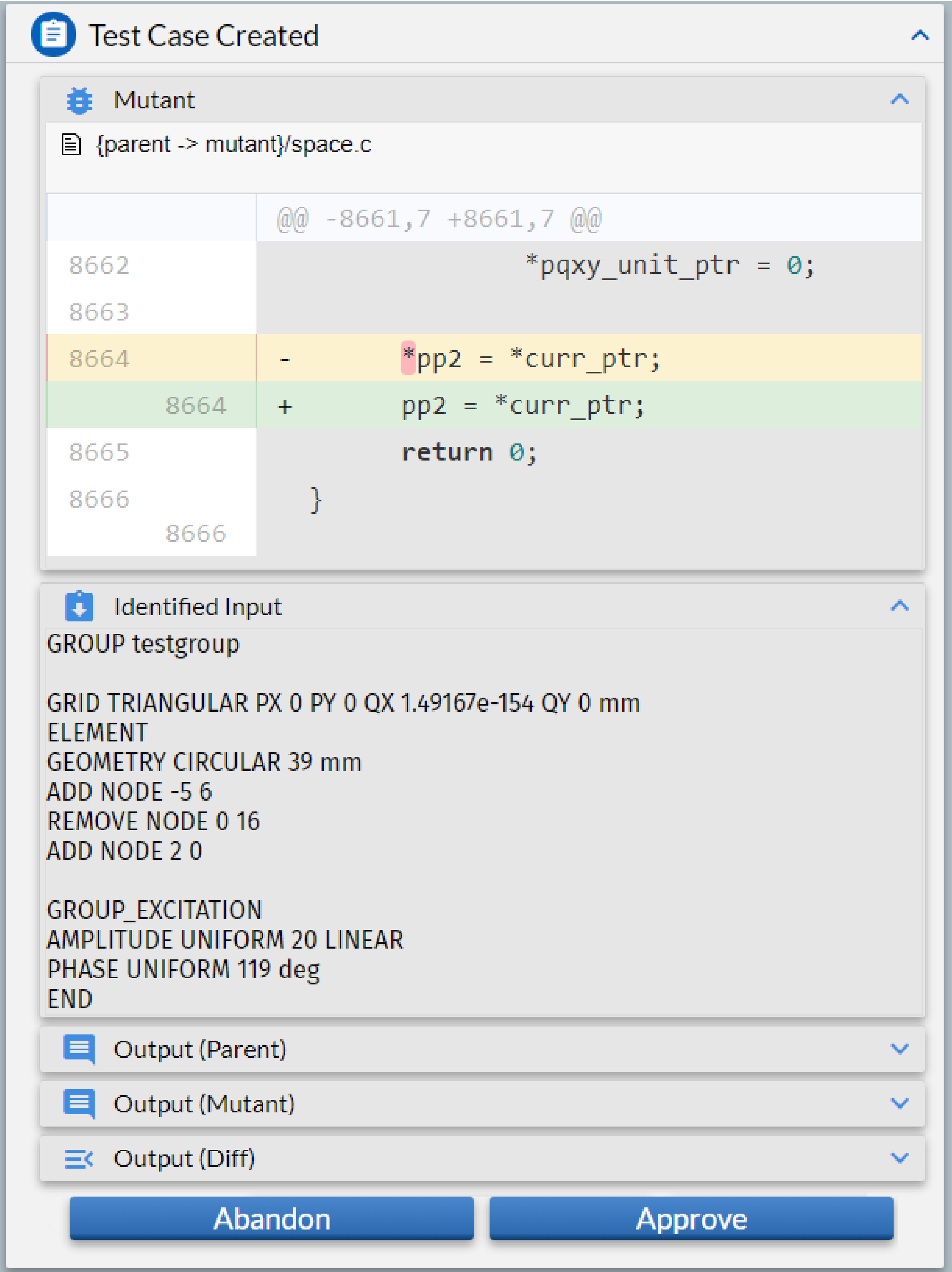
Our goal is to construct both individual test cases, as well as entire test suites. We accomplish these goals by finding live mutants, and then killing them through the application of target-oriented fuzzing. When our technique is applied to an individual mutant, it is capable of generating a new input to serve as a new test case. When applied to a corpus of mutants, our technique builds an entire test suite. In section 5.7, we present evidence that our technique creates an *effective* test suite, as measured by its *mutation-adequacy* score and its *code-coverage* score. Because prior research has shown that the mutation-adequacy score provides a reasonable approximation of the ability of a test suite to detect real defects (Andrews et al., 2005b), we believe our technique can be a powerful tool for an engineer seeking to build a stronger test suite for their project, either by adding individual test cases or generating an entire test suite.

Ultimately, our goal is to enable the development of a system for improving test suites quickly and with relatively low cost. Figure 5.1 shows an example of a web-based UI expressing our vision for this primary use of our technique, and the core outputs of the process:

- **A live mutant.**
- **An input killing the mutant.**
- **The differing outputs between the mutant and its parent program.** With the mutant, input, and difference in outputs, an engineer using the system can determine the value of the test case created by our technique.

We do not present our work as a panacea for test-suite development. Our technique supports the automatic generation of software-testing artifacts—both individual test cases and test cases aggregated as test suites—along with evidence for the value of those artifacts. We do, however, require existing (presumably)

Figure 5.1: Conceptual Web-Based Interface



human-authored test cases to start from, and expect some level of human supervision of the produced test cases, with the level of human involvement varying depending on the configuration of the tool-chain. Where we expect to benefit the greater software-engineering community is in the rapid expansion of test suites from a small, manually authored base to a larger, more robust test suite, while minimizing the human effort required to do so.

5.3 Definitions and Problem Statement

A test suite S for some program P is a set of test cases $\{C_0, C_1, ..., C_n\}$; each test case C_i consists of an input and output of the program: $C_i = \langle C_i^{in}, C_i^{out} \rangle$, where C_i^{in} is an input of P, and C_i^{out} is an output of P. A program P *passes* test case C_i if and only if $P[C_i^{in}] = C_i^{out}$. A program passes test suite S if and only if:

$$\forall C_i \in S : P[C_i^{in}] = C_i^{out} \tag{5.1}$$

We denote a program P passing a test suite S by $P \models S$. We assume that our technique is executed on a program in a state that passes all test cases; that is, for any program P and test suite S that we initially consider, we assume $P \models S$.

Assume that P is mutated in some way to generate a program P' that is not semantically identical to P. We can then execute the test suite to determine whether $P' \models S$ holds. Ideally, for an arbitrary mutation of the original program, the test suite should detect a difference in behavior. If this difference in behavior is detected—i.e., $P' \not\models S$—we say that mutant P' is *killed*. If the change to behavior is undetected—i.e., $P' \models S$—we say that P' is *live*.

Mutation testing (the technique used to generate P' above) is ordinarily used to evaluate the quality of a test suite. However, mutation testing *also* offers the opportunity to improve a test suite by uncovering live mutants that can be used to extend the program's test suite. To take advantage of this opportunity, we must kill the mutant.

Suppose that P' is a mutant that is live by S ($P' \models S$). Killing P' requires the

identification of a new "killing" test case C_{kill} so that $P \models \{C_{kill}\} \cup S$ and $P' \not\models \{C_{kill}\} \cup S$. Consequently, for C_{kill} to usefully kill P',

$$P[C^{in}_{kill}] = C^{out}_{kill} \wedge P'[C^{in}_{kill}] \neq C^{out}_{kill}, \tag{5.2}$$

or, more simply,

$$P[C^{in}_{kill}] \neq P'[C^{in}_{kill}]. \tag{5.3}$$

Thus, the problem that we address can be stated as follows: for a given program P and test suite S such that $P \models S$, identify an *argument* $A = \langle P', C^{in}_{kill} \rangle$, where P' is a mutant of P such that

$$P' \models S \wedge P[C^{in}_{kill}] \neq P'[C^{in}_{kill}]. \tag{5.4}$$

From A we can generate a new test case $C_{kill} = \langle C^{in}_{kill}, P[C^{in}_{kill}] \rangle$. If C_{kill} is accepted (that is, if the engineer using our tools accepts our argument), S is extended to $S' = \{C_{kill}\} \cup S$, and thus $P \models S'$ and $P' \not\models S'$.

Once A is determined, we present to the engineer both a live mutant and a test case that kills the mutant. We present this argument to the engineer as the following data:

- The mutation applied to generate P', that is, the alteration to the code that creates a difference in behavior that is not identified by the existing test suite S.
- The input C^{in}_{kill} itself.
- The outputs of both the original and mutated programs, that is, $P[C^{in}_{kill}]$ and $P'[C^{in}_{kill}]$.

In short, we give to the engineer a *plausible* change to the code that cannot be distinguished by the current test suite, and an addition to the test suite that can distinguish the change. The core concept of our work is that not only can we generate new test cases, but we can also generate *reasonable* arguments for adding these newly generated test cases to an existing test suite.

5.4 Technique Overview

Critical Points

Our technique depends on the computation of *critical points*. We define the critical points of two programs A and B as the set of nodes in the control-flow graph of the *first* program where its behavior diverges from the second program. That is, *critical_points*(A, B) is a set of nodes in the control-flow graph of A. Our input-generation tool works by steering program execution to reach the nodes in this set. This process is a key element of the technique, because for an input to differentiate the behavior of two programs (assuming that both programs are deterministic), the trace of the program executed on the input has to pass through at least one divergent basic block.

Our algorithm for determining critical points is focused on the analysis of *similar* programs—that is, programs with control-flow graphs expected to differ by only a small number of basic blocks. This assumption is reasonable because our target programs for the computation of critical points will always be a a program and its mutant, so the two programs analyzed will typically differ in two or fewer lines of source code.

For two similar programs A and B, we compute *critical_points*(A, B) in the following way:

We first apply a textual diff on the source code of A and B. The GNU diff program efficiently identifies where the two programs differ. The mutation system we use is configured to generate small changes (limited to 14 or fewer tokens, which results in the vast majority of mutations used causing changes in two lines of code or fewer). We can then map these lines in the source code of program A to the corresponding basic blocks in the program's control-flow graph. These nodes in the control-flow graph for program A are chosen as *critical_points*(A, B).

Input Generation

Once the critical points are computed, we use TOFU (which stands for Target-Oriented FUzzer) (Wang et al., 2020) to generate inputs that cause them to be executed. TOFU then fuzzes inputs, attempting to reach each basic block identified by *critical_points*(A, B) until it finds an input that causes programs A and B to behave differently. Note that TOFU's search is more than just to cause *critical_points*(A, B) to be executed: in addition, the difference in behavior must be detectable via a comparison of the programs' outputs.

With program P and mutant P′, once we have an input C^{in} that causes a particular node in *critical_points*(P, P') to be executed, we test that input for its potential as a test case—that is, whether it causes a difference in observable behavior: $P[C^{in}] \neq P'[C^{in}]$. If this input causes an observable difference in behavior in the programs, we use it as the basis for a new test case, and present it to the user as the argument $A = \langle P', C^{in}_{kill} \rangle$. If TOFU is unable to find an input that causes an observable difference in behavior, we continue to the next node in *critical_points*(P, P'), stopping execution once we have found a diverging input or reached a pre-determined timeout. The timeout, for our experiments, is implemented as a limit on the total number of iterations performed by TOFU as described in section 5.7.

5.5 Target-Oriented Fuzzing

Fuzzing

Fuzzing has been widely used in software testing. It is a technique for automated testing in which inputs are generated (often with some degree of randomness), and the program is executed on those inputs in the hope that some of the executions will trigger buggy behaviors of the subject program. One advantage of fuzzing is that when a bug is identified, an input that triggers it is also provided; the bug can be reproduced by rerunning the program using the input, and debugging

can start from a specific concrete execution.[1] Despite its seemingly too-simple methodology of running a program using randomized inputs, fuzzing is quite effective in identifying security vulnerabilities, and has become a standard technique for testing software (Godefroid, 2020).

In general, fuzzers can be categorized as belonging to one of three types: blackbox, whitebox, and greybox. Blackbox fuzzing assumes no knowledge of the source code, and tests the program with totally random inputs. In contrast, whitebox fuzzing assumes access to the source code, and uses heavyweight program-analysis techniques. The fuzzer symbolically executes the program, and collects constraints from the sequence of branch directions taken during the execution. To drive execution down a new path, the fuzzer obtains a path constraint π by (roughly) conjoining all of the branch-constraints for (some prefix of) the path, negating the final constraint, and then passes π to a constraint solver. If π is satisfiable, any satisfying assignment obtained from the constraint solver represents an input that drives execution down the prefix, and then *off* of the original path. Greybox fuzzing leverages some knowledge about the program; typically, lightweight program instrumentation is used to observe the state of an execution, and the information so obtained is used to guide input generation. The state-of-the-art fuzzer, American Fuzzy Lop (AFL), is a greybox fuzzer (Zalewski, 2020).

There are also different techniques that can be used for input generation. In designing a fuzzer, one consideration is whether a grammar will be used: a fuzzer can be either *grammar-aware* or *grammar-blind*. The grammar-aware approach generates the input with respect to a pre-specified grammar that captures the program's input language (or possibly some approximation to the input language). Because those inputs are either guaranteed to be valid, or are typically valid, an execution starting from such an input can dive deep into the subject program. However, the grammar-aware approach may requires manual effort to specify the grammar, and

[1]Reproducibility may be lost in the presence of nondeterminism—e.g., time-related behavior, or nondeterminism due to concurrency. In such situations, fuzzing can still be useful if programs are instrumented to provide data about the execution path followed, which can provide clues about bugs encountered during that (non-reproducible) run.

it is subject-specific.[2] A grammar-blind approach generates the input randomly without knowledge of the input format. However, most programs take inputs with certain format constraints. Most grammar-blind generated inputs are invalid, and cannot pass the input-validation stage of the subject program.

Another consideration is whether an input is generated from scratch or from an existing input: the input-generation approach can be either modification-based or from-scratch.[3] A modification-based approach generates an input from an existing input by changing the input a little—for example, by adding, deleting, or changing a few bits or bytes of an existing input. On the other hand, a from-scratch approach does not start from an existing input. In general, grammar-aware approaches have been from-scratch, while grammar-blind approaches have been modification-based (Godefroid, 2020).

Target-Oriented Fuzzing

While standard fuzzing aims to increase test coverage, it offers little control over which parts of the program are explored. However, in some cases, certain parts of the program are more interesting than others. In the mutant-killing scenario, when a certain part of a program is mutated, one is interested in generating an input that triggers a behavior different from the original program's behavior. Thus, a necessary condition is to generate an input that reaches the mutated part of the program. Consequently, one is interested in a *target-oriented fuzzer*, rather than a fuzzer that supports the standard fuzzing goal of increasing code coverage.

While synthesizing an input to reach some specific points in the program can be addressed using symbolic execution and a constraint solver, AFLGo has shown

[2]There has been some work on automatically generating a grammar for the language of a program's inputs (Hoschele and Zeller, 2017; Bastani et al., 2017; Wu et al., 2019).

[3]In work on fuzzing, the operation of modifying the input is often called "mutating the input." However, because in this chapter we have already used "mutating" to mean the process of introducing changes in the *program* (in the sense of mutation testing), we will use "mutating" when referring to changes introduced in a program, and "modifying" when referring to changes in an input. (The one exception to this terminology convention comes when discussing TOFU's modification-based input-generation component, which is based on libprotobuf-mutator (Google, 2020a).)

that, in practice, fuzzing—together with certain program-analysis techniques—is often an empirically better approach (Böhme et al., 2017). Given the target lines in the source code, the fuzzer needs to measure the “quality” of each input. If one input has an execution trace that gets closer to the target than another, then the first input should be considered better: mutating that input is more likely to produce inputs that get even closer, and eventually—after enough modifications to the inputs—reach the target. Consequently, the program analysis used in such a tool needs to provide a measure to decide the quality of each input. In essence, AFLGo is the combination of AFL and such an analysis, which computes a score for each basic block in the program to measure the distance from each basic block to the target.

AFLGo has some drawbacks as a target-oriented fuzzer because it inherits many features from AFL, which aims to increase coverage rather than reach a specific target. For example, AFL uses grammar-blind modification to generate inputs, many of which are invalid and rejected by the subject program. While unexpected invalid inputs can be useful for identifying security vulnerabilities and increasing coverage, they are less useful when the goal is to reach a specific target in the program—especially if the target is deep in the program, rather than in the input-parsing stage. For example, if we have an input whose execution trace is close to the target, structurally blind modification of some bits or bytes of the input is likely to make the input invalid and rejected by the program immediately. If one considers the topology induced by the grammar-blind modification operators, the neighboring inputs of the desired inputs are mostly invalid, and their executions are distant from the target. Consequently, structure-blind modification is less suitable for target-oriented fuzzing.

TOFU

In our tool-chain, to attempt to create test cases that kill mutants, we use TOFU. TOFU is a greybox, grammar-aware, modification-based fuzzer. Unlike traditional grammar-aware fuzzers, which generate inputs according to a grammar

from scratch, TOFU generates grammar-valid inputs by modifying existing inputs. Therefore, by examining existing inputs' execution traces, TOFU can choose the best input—ones whose execution traces are close to the target, and modify them according to the grammar, hoping that the new inputs have execution traces that get closer to the target, and eventually—with a sufficient number of modifications to the inputs—allow the target to be reached.

TOFU's modification-based input-generation component relies on Google's libprotobuf-mutator (Google, 2020a). Protocol-buffers ("protobufs") are a "language-neutral, platform-neutral extensible mechanism for serializing structured data" (Google, 2020b). Libprotobuf-mutator is used to randomly modify protobufs. When a user describes a grammar using the protobuf-specification language, the specification is compiled into a C++ class, and an input corresponds to an object in the class. The mutator can modify the object, by modifying sub-components to create a modified object according to the specification. The modified object corresponds to a modified input.

For each target line, TOFU first identifies the corresponding basic blocks in the control-flow graph, and then computes the distance from each basic block to each target basic block. TOFU uses a priority queue to store candidate inputs, and the priority is calculated from both the distances and the covered basic blocks induced by the input. Fuzzing is a repeated loop of modification (delegated to libprotobuf-mutator) and execution. During each iteration, TOFU selects a seed input, generates multiple new modified inputs, and then executes the program using the new inputs. The new inputs are then added to the priority queue according to their execution-distance-based priority. More details about TOFU can be found in (Wang et al., 2020).

5.6 Experimental Setup

Target Program

We used the Space program, part of the SIR repository (Do et al., 2005). This program was selected for multiple reasons. Space has a long history as a "model organism" in software-engineering research, appearing frequently in the literature for mutation testing (Just et al., 2014c; Andrews et al., 2005b).

In addition to being very well-known within the community, Space has a feature vital for our experiments—its 13 496-case test suite is massive. This feature is especially important for the validity of our experiments, because the detection of equivalent mutants is an open question that is beyond the scope of this chapter (an "equivalent mutant" is a mutant that is semantically equivalent to its parent program). Following the convention adopted by other researchers (Andrews et al., 2005b; Brown et al., 2017), we use this extensive test suite as a proxy for equivalence detection. That is, when computing mutation adequacy relative to Space's test suite, we assume that any mutant *not* killed by the test suite is an equivalent mutant. By removing such apparently equivalent mutants from consideration, we ensure that our tool-chain only tries to find differentiating inputs for mutants *known* to be non-equivalent.

Finally, Space is a relatively small program at 8718 lines of code. This feature makes it reasonable for our experiments to cover all possible mutants generated by the mutation-testing frameworks we used.

Mutant Generation

We used two different mutation-testing frameworks to generate mutants for our experiments. We used these frameworks both to maximize the total number of mutants available for experimentation, as well as to avoid potential biases in our results due to reliance on a single system or algorithm for mutating source code.

One of the core concepts of our technique is that we are able to furnish *arguments* that the test cases we generate are useful. To make these arguments more

comprehensible to a human engineer overseeing the process, we require that the mutations used generate readable differences in the source code of the program under test. Because of this requirement, we were unable to use frameworks that focus on the mutation of compiled artifacts, as opposed to source code.

In addition to requiring source-code-level mutation, we also preferred mutation-testing frameworks under recent development. Using these criteria, we chose to use the following mutation-testing frameworks:

- *SRCIROR* (Hariri and Shi, 2018)
- *Wild-caught mutants* (Brown et al., 2017).

For each mutation-testing framework used, we generated all possible compilable mutants of SPACE. In section 5.7, we describe how we used our tool-chain to create a new test-suite for SPACE, starting from just four randomly selected tests from the original SPACE test suite. In section 5.7, we describe how we used AFL (Zalewski, 2020) to generate a third test suite for SPACE. We evaluated each of the mutants for equivalence, using each test suite as a proxy for equivalence detection. Table 5.1 lists the results from mutant generation.

During the course of our experiments, we identified a set of mutants that exhibited non-deterministic behavior. Manual inspection of the entire set of mutants was not feasible, but inspection of a small subsample showed that some of these mutants could exhibit undefined behavior. We removed these mutants from our experimental corpus.

The result of this process left us with three corpora of mutants. In each, all mutants that exhibited non-deterministic behavior were removed. In addition, all mutants were removed that were considered to be equivalent by a given proxy for equivalence detection (SPACE, "Ours", and "AFL-generated"). We used the three corpora to carry out measurements of the effectiveness of the test suites relative to using SPACE, Ours, and AFL-generated as the proxy for equivalence (see section 5.7).

Table 5.1: Mutation-testing frameworks

Mutation-testing framework	Mutants			
			Relative to SPACE[a]	
	Generated	Non-deterministic	Considered equivalent	Experimental corpus
SRCIROR	11 374	928	3459	6987
Wild-caught mutants	27 641	1381	9522	16 738
Total	39 015	2309	12 981	23 725

Mutation-testing framework	Mutants			
	Relative to Ours[b]		Relative to AFL[c]	
	Considered equivalent	Experimental corpus	Considered equivalent	Experimental corpus
SRCIROR	3961	6485	6609	3837
Wild-caught mutants	9946	16 314	17 133	9127
Total	13 907	22 799	23 742	12 964

TOFU

As described in section 5.5, TOFU implements target-oriented fuzzing through a repeated loop that involves (i) input generation by modifying the input that is currently estimated to be of "highest quality," (ii) execution of an instrumented version of the program on the modified inputs, and (iii) classification of the quality of the results (based on the shortest distance reached during execution to one of the target blocks). In our experiments, TOFU generates 80 new inputs on each iteration (i.e., each time step (i) is performed). In the different experiments described in section 5.7, we vary both the iteration limit and the number of seed inputs.

TOFU is used to generate inputs that reach specific target basic blocks in the subject program (so-called "critical points"). However, reaching a target basic block is necessary but not sufficient to kill a mutant; for the mutant to be killed,

there must be a difference in *observable* behavior between the original program and the mutant. Because our interest is in generating test cases, a difference must show up in the *outputs* of the original program and the mutant. Depending on the nature of the mutation, the program that contains the critical points could be either the original program or the mutant; call that program P. TOFU is used to fuzz P until it generates an input that reaches a target basic block. At that point, both programs are run, and their respective outputs in both stdout and stderr are compared. If there are any differences, then TOFU has generated an input that leads to a behavioral difference in the original program and the mutant.

5.7 Experiments and Results

Research Question 1

RESEARCH QUESTION 1: Can target-oriented fuzzing be used to create test cases to kill live mutants?

To answer this research question, we used TOFU to generate inputs. As described in section 5.3, each of these inputs, when combined with output recorded from the program under test, represents a new test case. For each mutant we generated, we used our technique to attempt to construct a new test case to kill it. To evaluate the utility of our technique across a range of levels of resource availability to an engineer using the system, we evaluated our technique's ability to construct new test cases along two dimensions:

Iterations. For each experimental run of TOFU, we configured it to stop execution after a fixed number of iterations. Because the high-throughput computing platform we used for our experiments lacks reliable "wall-clock" timing, we used iterations of TOFU's target-oriented-fuzzing algorithm as a proxy for processor-time used. To be able to report on the ability of our technique to generate test cases across a wide range of available computing resources, we ran TOFU with limits of 200, 500, and 5000 iterations.

Table 5.2: Test-case generation and TOFU execution times

Success Rate			
	Seed Test Cases		
Iterations	5	20	100
200	83%	83%	85%
500	84%	87%	87%
5000	85%	87%	91%

(a) Test-case generation results. Each entry is the percentage of mutants in the corpus for which TOFU was able to create a test case that killed the mutant, as a function of the number of seed test cases and number of iterations performed by TOFU. (A higher percentage indicates a greater ability to create test cases.)

Median Execution Time			
	Seed Test Cases		
Iterations	5	20	100
200	2s	5s	5s
500	2s	5s	5s
5000	2s	5s	5s

Maximum Execution Time			
	Seed Test Cases		
Iterations	5	20	100
200	1m 2s	1m 4s	57s
500	2m 18s	2m 19s	2m 45s
5000	20m 48s	22m 56s	22m 38s

(b) Single-mutant TOFU execution times. Each entry shows the *median* and *maximum* time observed running TOFU on a set of mutants sampled from the experimental set. Maximum here represents TOFU “timing out” due to it hitting the specified iteration limit. These executions were performed on a desktop computer with an Intel® Core™ i7-9700k CPU and 32 GB of memory.

Seed test cases. While TOFU *can* create inputs from thin air, the target-oriented-fuzzing process is more effective when seed inputs are provided. Our goal is for this technique to be usable in projects with both large and small test suites—both to build up a test suite rapidly from a small basis, and to plug holes in an already-robust test suite. To evaluate our technique's performance under these conditions, we attempted to generate test cases using 5, 20, and 100 randomly selected pre-existing test cases as a starting basis. Then, for each mutant in our set of non-equivalent mutants, we ran TOFU on the original program and the mutant (in hopes of finding an input differentiating the two programs) with the set of seed test cases.

In all executions of TOFU to generate new test cases, we used seed test cases that were randomly sampled from Space's test suite. Because we used Space's test suite as a proxy for equivalence detection, for all of these mutants at least one test case already existed that killed the mutant; the test cases used for input here were sampled exclusively from test cases that did **not** kill the mutant being analyzed. The seed test cases used for each run of TOFU were sampled independently.

Our results are summarized in table 5.2(A). Each entry is the percentage of mutants in the corpus for which TOFU was able to create a test case that killed the mutant, as a function of the number of seed test cases and number of iterations performed by TOFU. (A higher percentage indicates a greater ability to create test cases.) With success rates of test-case generation ranging from 83% to 91%, we answer this research question in the affirmative. The identification of critical points, followed by target-oriented fuzzing can be used to kill live mutants.

Table 5.2(B) describes the execution time of TOFU taken from a sample of the entire mutation set. *Most* of TOFU's executions completed quickly—providing the user with an input differentiating the mutant from its parent program—but many "timed out" by hitting the provided iteration limit. We found that only the number of test cases provided as seed input to the program had any impact on the TOFU's median execution time. Seed-input count had little impact on TOFU's maximum-execution time, but changing the iteration limit produced a nearly linear increase in maximum execution time. Both of these observations are in line with our test-case-generation results. Even with a limit of 200 iterations, TOFU is capable

of finding differentiating inputs for most mutants, because of this the iteration limit will not affect running time for the *median* mutant. In the worst case, the execution of TOFU times out, and the time taken to do so is roughly linear with respect to the iteration limit.

Research Question 2

> RESEARCH QUESTION 2: Can target-oriented fuzzing combined with mutation analysis be used to generate robust test suites?

To answer this research question, we extended our test-case-generation process to construct an entire test suite, as opposed to sampling the technique's ability to kill mutants in isolation. This extension of the technique allows us to evaluate our ability to generate entire test suites from minimal starting inputs. We performed this experiment for each of the three different iteration-limit configurations used for the test-case-generation experiment, and report the results of these trials.

We selected four test cases randomly from SPACE's 13 496-case test suite to use as an initial test suite. We used these four test cases as a set of seed inputs to attempt to generate a mutant-killing input for each mutant not already killed by the seed inputs. Those generated inputs that killed mutants were added to the set of seed inputs (as well as the test suite under construction), and killed mutants were marked as such. Once these inputs were collected, we returned to the mutants *not* killed by the previous iteration. We then attempted to kill the previously live mutants with the now-larger set of seed inputs produced by adding the newly-generated inputs from the previous iteration to the test suite for the current iteration. This process—attempting to kill live mutants in batches, adding to the seed-input set each iteration, in order to maximize parallelization—was repeated until a full iteration was performed on the set of live mutants without killing any additional mutants. Because we increase the size of the test suite (and hence the seed-input set) whenever we are able to construct a new test case, the more mutants we kill, the more likely the test suite being constructed is to kill future mutants.

After processing each mutant this way, we restart the process on the set of (still) live mutants, attempting to find inputs for each of them using the growing test suite as a basis for target-oriented fuzzing. This process continues until we either kill all mutants or make a full pass through the live-mutants set without constructing any new test cases. The resulting test suite after this step is our "comprehensive" test suite.

During the evaluation of our test suites, we observed that our synthetic test suites required substantially more execution time than Space's human-authored suite. Space itself allows test cases to produce arbitrarily large outputs because its inputs are, effectively, programs with the functionality to iterate over ranges specified in the inputs. The human-authored test cases tend to avoid abusing this functionality: the largest output from the original test suite was roughly 250 kB. TOFU, however, generated *some* inputs whose outputs were greater than 600 MB. Because we would expect the typical engineer using our technique to find test cases with such massive outputs to be unusably large, we created "filtered" versions of our test suites by removing the roughly 7% of test cases for which the output size was over 1 MB—i.e., those test cases whose output size was greater than four times the size of the largest output from Space's own test suite. After these test cases were removed, our synthetic test suites had comparable metrics for both median test-case output size and total execution time.

Because our experimental protocol uses Space's test suite as a proxy for the detection of equivalent mutants, any mutation-adequacy score we report for our test suite(s) will be relative to the mutation-adequacy score of the Space test suite itself. In addition to this mutation-adequacy score, we also computed the mutation-adequacy score of Space's test suite in relation to our test suites—that is, we treated *our* test suite as the proxy for equivalence detection, and calculated the mutation-adequacy score of Space's test suite against the set of mutants our test suite was able to kill. Figure 5.2 provides a conceptual view of this relationship; for any set of mutants, some will be "inequivalent" relative to Space's test suite, and some inequivalent relative to our test suite. The intersection of the two regions represents those mutants inequivalent—that is, detected by—both test suites.

Table 5.3: Test-suite Metrics

Test Suite	Size	Time to Generate	Basic-Block Coverage	Relative Mutation Adequacy: Relative to SPACE[a]	Relative to Ours[b]	Relative to AFL[c]
"Controls"						
SPACE's	13 496	N/A	89%	100%	96%	>99%
AFL-generated	35 296	9200 hours[d]	66%	54%	56%	100%
Ours (200-iters.)						
Comprehensive	20 545	400 hours[e]	85%	87%	93%	>99%
Filtered	19 429	N/A[f]	85%	83%	89%	97%
Ours (500-iters.)						
Comprehensive	20 664	800 hours[e]	85%	88%	93%	>99%
Filtered	19 485	N/A[f]	85%	84%	90%	97%
Ours (5000-iters.)						
Comprehensive	21 969	6000 hours[e]	86%	93%	100%	>99%
Filtered	20 233	N/A[f]	85%	87%	94%	98%

[a] Mutation-adequacy scores relative to SPACE's test suite. Failure of SPACE's test suite to kill a mutant is used as a proxy for mutant-equivalence. By definition, the relative mutation-adequacy score for SPACE's test suite is 100%.
[b] Mutation-adequacy scores relative to our best constructed test suite, specifically, the one generated with TOFU's timeout set to 5000 iterations. For this column, failure of our test suite to kill a mutant is used as a proxy for mutant-equivalence. By definition, the relative mutation-adequacy score for the "Comprehensive" test suite of the 5000-iteration version is 100%.
[c] Mutation-adequacy scores relative to the test suite generated by AFL. For this column, failure of this test suite to kill a mutant is used as a proxy for mutant-equivalence. By definition, the relative mutation-adequacy score for AFL's test suite is 100%.
[d] Starting with four inputs on server-grade hardware, see section 5.7. Normalized to single-core hours.
[e] Starting with four inputs on a high-throughput computing platform. Due to unreliable measurement of "wall-clock" time, these values are approximate.
[f] Generation of the "Filtered" test suite has no effective cost beyond the generation of the "Comprehensive" test suite, because it is constructed from meta-data collected during the process.

Figure 5.2: Relative Non-equivalence

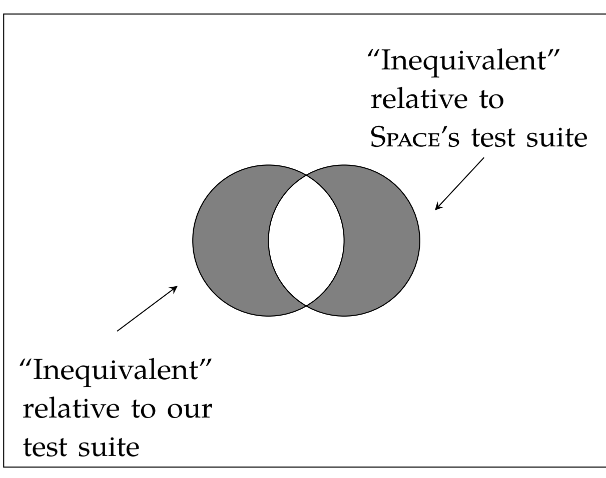


Table 5.4 summarizes our results from this test-suite-generation process, comparing our test suites to both Space's test suite and an AFL-generated test suite as a second "control".

Because equivalence detection is undecidable, any proxy used to approximate it will be imperfect. We compare the quality of the proxies used in our experiments—Space's test suite, our best test suite, and the AFL-generated test suite—in table 5.5. Notably, we were able to kill 804 mutants live by even Space's test suite.

Because we were able to generate test suites with high mutation adequacy (and even kill some mutants that managed to escape Space's expansive test suite), we answer this research question in the affirmative.

Research Question 3

> Research Question 3: How do these generated test suites compare to manually-authored test suites in size, code coverage, and the ability to kill mutants?

To more effectively evaluate the quality of our test suites, in addition to mutation-adequacy score, we also computed the code coverage of the test suites generated

Table 5.4: Test-suite Metrics

Test Suite	Size	Time to Generate	Basic-Block Coverage	Relative Mutation Adequacy: Relative to SPACE	Relative to Ours	Relative to AFL
"Controls"						
SPACE's	13 496	N/A	89%	100%	96%	>99%
AFL-generated	35 296	9200 hours	66%	54%	56%	100%
Ours (200-iters.)						
Comprehensive	20 545	400 hours	85%	87%	93%	>99%
Ours (500-iters.)						
Comprehensive	20 664	800 hours	85%	88%	93%	>99%
Ours (5000-iters.)						
Comprehensive	21 969	6000 hours	86%	93%	100%	>99%

[a] Mutation-adequacy scores relative to SPACE's test suite. Failure of SPACE's test suite to kill a mutant is used as a proxy for mutant-equivalence. By definition, the relative mutation-adequacy score for SPACE's test suite is 100%.
[b] Mutation-adequacy scores relative to our best constructed test suite, specifically, the one generated with TOFU's timeout set to 5000 iterations. For this column, failure of our test suite to kill a mutant is used as a proxy for mutant-equivalence. By definition, the relative mutation-adequacy score for the "Comprehensive" test suite of the 5000-iteration version is 100%.
[c] Mutation-adequacy scores relative to the test suite generated by AFL. For this column, failure of this test suite to kill a mutant is used as a proxy for mutant-equivalence. By definition, the relative mutation-adequacy score for AFL's test suite is 100%.
[d] Starting with four inputs on server-grade hardware, see section 5.7. Normalized to single-core hours.
[e] Starting with four inputs on a high-throughput computing platform. Due to unreliable measurement of "wall-clock" time, these values are approximate.
[f] Generation of the "Filtered" test suite has no effective cost beyond the generation of the "Comprehensive" test suite, because it is constructed from meta-data collected during the process.

Table 5.5: Quality of various proxies for mutant equivalence

Test suite used as equivalence proxy	# Mutants considered equivalent (i.e., not killed by proxy)	# of "Equivalent" Mutants Killed by		
		SPACE's	Ours (5000-iter.)	AFL-generated
SPACE's	12 981	0	804	81
Ours (5000-iter.)	13 907	1730	0	118
AFL-generated	23 742	10 842	9953	0

by our technique. To determine code coverage, we used TOFU's instrumentation system to trace basic-block execution, and evaluated our results based on the percentage of the whole program's basic blocks that were executed by the test suites being examined.

To provide a point of reference for the code coverage achieved by our synthesized test suite, we compared our results to SPACE's expansive human-authored test suite. SPACE's test suite is well-known for its expansiveness, containing more test cases than the program itself has lines of code.

As an additional "control" for our experiments, we also created a test suite automatically using AFL. AFL is a state-of-the-art fuzzing tool that tries to identify the buggy behavior of programs by increasing code coverage (Zalewski, 2020). We ran AFL in parallel mode on a workstation with twenty-four Intel® Xeon® X5675 CPUs running at 3.07 GHz and 189 GB of memory for 16 days starting with the same four seed test cases used during our test-suite-generation experiment. After allowing AFL to run for 16 days, we collected the 35 296 inputs it generated as a test suite.

The size, code coverage, and mutation-adequacy scores of all of the analyzed test suites—SPACE's human-authored test suite, our generated test suites, and AFL's synthesized test suite—are summarized in table 5.4.

Our technique was able to generate effective test suites achieving a mutation-

adequacy score of 87% (relative to Space's test suite) and 85% basic-block coverage (this coverage amounts to 96% of the basic-block coverage of Space's test suite) in our test suite generated with a computational limit of 200 iterations of TOFU. The technique was able to achieve a mutation-adequacy score of 93%, killing roughly half of the remaining mutants, with the 5000-iteration-limit version of the test suite. The computational efficiency of the technique drops off sharply from the 200-iteration suite to the 5000-iteration suite, the 5000-iteration version requiring approximately ten times the computational power to kill 5% more mutants than the 200-iteration version. However, our tool-chain is designed to give an engineer using the system the ability to choose how much computational power to devote to the process.

While the test suites generated automatically via our tool-chain do not outperform Space's test suite in mutation-adequacy score or code coverage, they approach it in both metrics, while requiring minimal engineering time dedicated to test-suite creation. The test suites generated automatically via our tool-chain surpassed the one created using AFL in both mutation-adequacy score and code coverage, while using broadly similar amounts of computing power.

Discussion

We were able to generate effective test suites, albeit at the cost of thousands of hours of processor time. This cost is not unique to our technique; mutation testing as a technique in software engineering has suffered from slow uptake within the industry due to this computational cost. Mitigating this cost through the use of cloud-based systems is not a new concept (Saleh and Nagi, 2015), and along similar lines we expect that the primary use case of our technique will involve cloud computing. We acknowledge that our technique is computationally expensive, but we believe it offers a compelling alternative to less-automated methods of test-suite construction.

Even though our technique incurs a high cost in computational power, the actual cost is small compared to the alternative. The median salary of a software developer

in the United States translates, roughly, to a wage of $50 USD per hour (Bureau of Labor Statistics, 2020) and is trending upwards. Non-time-critical computing with capacity most-similar to our experimental hardware[4] is available from Amazon at less than $0.04 USD per hour (Amazon, 2020), with the overall cost trending downward. While expensive in terms of raw computing resources, our technique allows the creation of robust test suites—containing many thousands of test cases—for prices commensurate with a day of a software engineer's work. Because the output of our tool-chain is comparable to the work of engineers, we believe that this comparison is a reasonable one to make.

Moreover, we believe that the *results* of our technique provide a compelling argument for its value. In contrast to typical mutation-testing systems, the technique provides *immediately usable* software-engineering artifacts in the form of test cases, as opposed to just an evaluation of the *quality* of a test suite, which is the normal output of mutation testing.

It is difficult to quantify the cost of test-case development by humans, especially because for a project like Space the test suite was developed over a considerable period of time, and has contributions from many developers. To get an estimate for the engineering resources required to recreate Space's test suite (or a test suite of similar size and content), we polled five software engineers, all with between ten and twenty-five years of professional experience. We described the Space program, the kind of test cases, and the size of the test suite, and asked for rough estimates of the engineering resources, in terms of man-hours spent by engineers, required to generate a test suite of comparable size. The responses from this question ranged from two to eighteen man-months to reproduce the test suite, with a median of six man-months.

Assuming that the median value of six man-months is a reasonable estimate, the salary alone for a developer to construct the test suite costs $50 000 USD. With current cloud-computing costs and the processing time used during the creation of our test suite, we were able to generate a test suite with 87% of the mutation-

[4]Based on requested memory size, the nodes we requested from our high-throughput computing platform are most similar to Amazon's "a1.2xlarge" level of service.

adequacy score, 96% of the code coverage, for less than 1% of the dollar cost.

5.8 Threats to Validity

Internal

A prominent threat to the validity of our experiments is our handling of equivalent mutants. The detection of equivalent mutants, that is, those mutants that are *semantically identical* to their parent program, is a long-standing problem in the field of mutation testing (Offutt and Lee, 1991; Grün et al., 2009). Active research continues into the problem (Ayad et al., 2019), and a solution to the problem (at least, in its more general form) is outside of the scope of this work; we avoid the problem of equivalent mutants creating false negatives in our experiments (that is, our system being unable to generate a test case distinguishing a mutant from its parent because the mutant is semantically identical to its parent) by explicitly computing our mutation-adequacy scores relative to proxies for equivalence detection. Although imperfect, this technique allows us to limit our experimental corpora to sets of mutants *known* not to be equivalent mutants. Prior work on mutation testing has used SPACE's test suite as a proxy for equivalence detection (Andrews et al., 2005b); we go beyond that work by using three different test suites—SPACE's, our best one, and one created using AFL—as such proxies. In particular, table 5.4 reports mutation-adequacy scores computed relative to each of the test suites.

Moreover, the results given in table 5.5 provide an "internal sanity check" of the hypothesis that a large test suite provides a reasonable approximation of an equivalence test. Our best test suite was able to kill relatively few mutants that were live by SPACE's test suite, and SPACE's test suite was able to kill relatively few mutants that were live by our best test suite.

Mutation adequacy has been shown to be a reasonable approximation of test-suite quality (Andrews et al., 2005b).

Finally, the comparison between our test suites and the test suite created using AFL supports the claim that our technique is able to create *robust* test suites—*i.e.,*

the test suites created via our technique have characteristics more similar to Space's test suite than the test suite created using AFL.

External

The most obvious external threat to validity of our work is the sheer computational cost of the technique, which we discuss in detail in section 5.7.

5.9 Related Work

Test-Case Generation

Test-case generation is an active field of research, with a wide variety of techniques under continuous development. While considerable research in the field is focused on the construction of test cases from models or requirements (Li et al., 2017), or specifically targeting code-coverage metrics (Yang et al., 2009), here we review related work instead on test-case-generation research that focuses on particular qualities of the test cases generated. (We consider the most-interesting element of our work to be that our generated test cases have an important feature—they distinguish behavior between the program under test and a *plausible* modification to it.)

Panichella et al. (2017) discuss an algorithm for improving the quality of test-case generation, using multiple metrics, including mutation-adequacy score ("strong mutation coverage" in their terminology). They use a genetic algorithm, as opposed to our target-oriented-fuzzing technique, as well as considerably smaller mutant sets for analysis.

Palomba et al. (2016) discuss test-code quality, arguing for the importance of the consideration of maintenance and execution costs of test suites, and mitigating these costs by preferring to use higher-quality test cases. Our work has a similar focus—the generation of test cases along with arguments for their relevance—but instead

of focusing on measurement of code quality, our work is focused on generation of the test cases themselves.

Xin and Reiss (2017) demonstrate a technique for detecting patches that are overfitted relative to a program's test suite—that is, a patch that "passes the test suite but does not actually repair the bug"—and generating test cases that make similar overfitting easier to detect in the future. Their work has a focus similar to ours, using these patches in place of the mutants that we use, but lacks the target-oriented-fuzzing element and the focus on large-scale test-case-generation.

Target-Oriented Fuzzing

Target-oriented fuzzing was first introduced by AFLGo (Böhme et al., 2017) under the name "directed fuzzing." Unlike standard fuzzing, which has the goal of maximizing the coverage of a test suite, the goal of target-oriented fuzzing is to generate inputs that cause the execution of specific locations in the source program's control-flow graph. Existing applications of target-oriented fuzzing include patch testing and crash reproduction, where the interesting parts in the program are newly-patched code or sections of the code that are known to cause a crash. In the mutant-killing setting, the interesting parts of the program are the mutated code. TOFU refines AFLGo by introducing distance-based metrics and a grammar-aware mutator, enabling TOFU to generate inputs faster.

Other than fuzzing, symbolic execution is also commonly used for program testing. Similar to standard fuzzing, the goal of symbolic execution is to maximize test-suite coverage. Guided symbolic execution was also proposed to generate inputs to reach specific locations during program execution (Marinescu and Cadar, 2013; Ma et al., 2011). However, Böhme et al. (2017) found that, in practice, the guided-symbolic-execution approach is not as efficient as the target-oriented-fuzzing approach.

5.10 Conclusion

In this chapter, we present a tool-chain that allows a software engineer either to improve the quality of an existing test suite by generating new test cases (along with evidence of each test case's utility), or to create an entire test suite, starting from only a handful of manually-created test cases. Our experiments provide evidence that a test suite so constructed has similar performance characteristics to a hand-constructed test suite (measured by mutation-adequacy scores and code-coverage scores). Moreover, our tool-chain can create such a test suite for a fraction of the cost (in time and dollars).

6 CONCLUSION

This chapter summarizes my doctoral research and presents a brief discussion of future research directions.

6.1 Mutation Testing

Ultimately, software testing is the most-effective bulwark against software failure currently known. My work has been primarily focused on improving techniques in mutation testing with the goal of improving testing itself.

For mutation testing to provide a useful measure of the adequacy of a test suite, it must produce not only faults within the system under test, but faults that mimic those caused by the actual developers working on a project. Just et al. demonstrated that faults introduced through mutation testing can serve as proxies for real faults introduced by developers and be effectively used to evaluate the sensitivity of a testing suite, although they also described limitations of existing sets of mutation operators.

My work has expanded upon the existing state of the art by the introduction of a technique—"*wild-caught mutants*"—allowing the automatic harvesting of new mutation operators. As opposed to existing "synthetic" mutation-testing techniques, every mutation created by this technique is based on a change that some real programmer made to some real piece of code.

Mutation testing, while yielding obvious benefits, has not yet found widespread use within the software-engineering community. Exhaustive approaches to mutation testing continue to be too computationally intensive to see wide adoption (Petrovic et al., 2018), with random sampling of mutants being one of the most effective methods used to reduce this computational cost (Gopinath et al., 2016). If mutation testing is to be adopted by the industry at large, it will need to conform to *reasonable* limits on its computational cost.

My work in *guided mutation testing* presents a technique to alleviate this fundamental problem. I describe a technique for improving the efficiency of mutation testing through the prioritization of mutants. Because the *generation* of mutants requires vastly fewer resources than their evaluation, my work on the prediction of various qualities of mutants has the potential to reduce the total computational cost associated with mutation testing while prioritizing more "interesting" mutants.

Finally, my work with *test-case generation through target-oriented fuzzing* offers a

new technique to synthesize test cases rapidly through an extension of mutation-testing techniques. I present a tool-chain that allows a software engineer either to improve the quality of an existing test suite by generating new test cases (along with evidence of each test case's utility), or to create an entire test suite, starting from only a handful of manually-created test cases. My experiments provide evidence that a test suite so constructed has similar performance characteristics to a manually-constructed test suite (measured by mutation-adequacy scores and code-coverage scores). Moreover, this new tool-chain can create such a test suite for a fraction of the cost (in time and dollars) of the manually-constructed one.

6.2 Future Work

While my work has been successful in introducing new techniques in mutation testing, time is, of course, a finite resource. Many opportunities to expand on my current work exist.

Mutation operators

My "wild-caught mutants" work provides techniques to generate new mutation operators, but my experiments have exposed areas for improvement.

Primarily among these areas is the cost of compilation; the wild-caught-mutants technique suffers from a low compilation rate compared to other mutation-testing techniques. The technique could be improved through tighter integration with the compilation process—to weed out uncompilable mutants earlier in the process. While a primary goal of the original tool-chain was to generate tooling as close to language-agnostic as possible (that is, to develop a tool-chain that could work with any nearly any programming language), stronger integration with build tools of individual languages would improve the utility of the technique.

Test-case analysis

While I think the results from my test-case generation work were particularly exciting, there is ample room to extend the technique.

Conceptually, a test suite provides a "sketch" of program behavior, and not all test cases contained within a test suite are of equal value to this representation of behavior. My technique was built around the concept of providing an *argument* to the engineer using it—that is, being able to provide not only an input and output defining the test case itself, but additionally a plausible modification to the program creating a *need* for the test case. This argument, however, does not guarantee the usefulness of a generated test case, nor the acceptance of the test case by an engineer overseeing the process.

My conception of the next step of this process is the analysis of what and *why* test cases are acceptable to a human engineer. I believe that machine-learning techniques could be used to compare test cases accepted or rejected by human engineers, and from this analysis produce models prioritizing automatically-constructed test cases, thus improving the quality of cases generated and presented for review by the engineers using the system.

REFERENCES

Amazon. 2020. Amazon ec2 spot instances pricing. `https://aws.amazon.com/ec2/spot/pricing/`.

Andrews, J. H., L. C. Briand, and Y. Labiche. 2005a. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on software engineering*, 402–411. ICSE '05, New York, NY, USA: ACM.

Andrews, James H, Lionel C Briand, and Yvan Labiche. 2005b. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on software engineering*, 402–411.

Ayad, Amani, Imen Marsit, JiMeng Loh, Mohamed Nazih Omri, and Ali Mili. 2019. Estimating the number of equivalent mutants. In *2019 ieee international conference on software testing, verification and validation workshops (icstw)*, 112–121. IEEE.

Bastani, Osbert, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. *SIGPLAN Not.* 52(6):95–110.

Böhme, Marcel, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 acm sigsac conference on computer and communications security*, 2329–2344. CCS '17, New York, NY, USA: ACM.

Boogerd, Cathal, and Leon Moonen. 2008. Assessing the value of coding standards: An empirical study. In *24th IEEE international conference on software maintenance (ICSM 2008), september 28 - october 4, 2008*, 277–286. Beijing, China: IEEE Computer Society.

Brown, David Bingham, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 511–522. ACM.

Budd, Timothy A., and Dana Angluin. 1982. Two notions of correctness and their relation to testing. *Acta Informatica* 18(1):31–45.

Bugzilla development team. 2016. Home :: Bugzilla :: bugzilla.org.

Chollet, François, et al. 2015. Keras. `https://keras.io`.

Coles, Henry. 2017. Pit mutation testing. [Online; accessed Jun. 2017].

Čubranić, Davor, and Gail C. Murphy. 2003. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th international conference on software engineering*, 408–418. ICSE '03, Washington, DC, USA: IEEE Computer Society.

Demillo, Richard, R.J. Lipton, and F.G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11:34 – 41.

DeMillo, Richard A., Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 11(4): 34–41.

Do, Hyunsook, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal* 10(4): 405–435.

Fischer, Michael, Martin Pinzger, and Harald Gall. 2003. Populating a release history database from version control and bug tracking systems. In *Proceedings of the international conference on software maintenance*, 23–. ICSM '03, Washington, DC, USA: IEEE Computer Society.

Godefroid, Patrice. 2020. Fuzzing: Hack, art, and science. *Commun. ACM* 63(2): 70–76.

Google. 2020a. libprotobuf-mutator.

———. 2020b. Protocol buffers.

Gopinath, Rahul, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. 2016. On the limits of mutation reduction strategies. In *2016 ieee/acm 38th international conference on software engineering (icse)*, 511–522. IEEE.

Grün, Bernhard JM, David Schuler, and Andreas Zeller. 2009. The impact of equivalent mutants. In *2009 international conference on software testing, verification, and validation workshops*, 192–199. IEEE.

Hamlet, Richard G. 1977. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.* 3(4):279–290.

Hariri, Farah, and August Shi. 2018. Srciror: A toolset for mutation testing of c source code and llvm intermediate representation. In *Proceedings of the 33rd acm/ieee international conference on automated software engineering*, 860–863.

Hellendoorn, Vincent J, and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?

Hellendoorn, Vincent J, Premkumar T Devanbu, Oleksandr Polozov, and Mark Marron. 2019. Are my invariants valid? a learning approach. *arXiv preprint arXiv:1903.06089*.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9(8):1735–1780.

Hoschele, M., and A. Zeller. 2017. Mining input grammars with autogram. In *2017 ieee/acm 39th international conference on software engineering companion (icse-c)*, 31–34.

Inozemtseva, Laura, and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*, 435–445.

Jia, Yue, and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37(5):649–678.

Just, R., D. Jalali, and M.D. Ernst. 2014a. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Int. symp. on softw. testing and analysis*.

Just, René. 2014. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 international symposium on software testing and analysis*, 433–436. ISSTA 2014, New York, NY, USA: ACM.

Just, René, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014b. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, 654–665. FSE 2014, New York, NY, USA: ACM.

Just, René, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014c. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering*, 654–665.

Kim, Sunghun, and Michael D. Ernst. 2007. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the european software engineering conference and the acm sigsoft symposium on the foundations of software engineering*, 45–54. ESEC-FSE '07, New York, NY, USA: ACM.

Kurtz, Bob, Paul Ammann, Jeff Offutt, Márcio E Delamaro, Mariet Kurtz, and Nida Gökçe. 2016. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 571–582. ACM.

Bureau of Labor Statistics, U.S. Department of Labor. 2020. Occupational outlook handbook. `https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm`.

Lawall, Julia, Ben Laurie, René Rydhof Hansen, Nicolas Palix, and Gilles Muller. 2010. Finding error handling bugs in openssl using coccinelle. In *Proceeding of the 8th european dependable computing conference, edcc 2010*, 191–196. Valencia, Spain: IEEE Computer Society.

Lawall, Julia L., Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. 2009. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *The 39th annual ieee/ifip international conference on dependable systems and networks*, 43–52. Estoril, Portugal: IEEE Computer Society.

Le, Xuan-Bach D., David Lo, and Claire Le Goues. 2016. History driven program repair. In *Software analysis, evolution, and reengineering (saner), 2016 ieee 23rd international conference on*, vol. 1, 213–224. Suita, Osaka, Japan: IEEE Computer Society.

Li, Wenbin, Franck Le Gall, and Naum Spaseski. 2017. A survey on model-based testing tools for test case generation. In *International conference on tools and methods for program analysis*, 77–89. Springer.

Ma, Kin-Keung, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed symbolic execution. In *Static analysis*, ed. Eran Yahav, 95–111. Berlin, Heidelberg: Springer Berlin Heidelberg.

Marinescu, Paul Dan, and Cristian Cadar. 2013. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 235–245. ESEC/FSE 2013, New York, NY, USA: ACM.

Matsumoto, Makoto, and Takuji Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.* 8(1):3–30.

memcached community. 2017. Memcached.

Merkel, Dirk. 2014. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal* 2014(239):76–91.

Mockus, Audris, and Lawrence G. Votta. 2000. Identifying reasons for software changes using historic databases. In *Proceedings of the international conference on software maintenance (icsm'00)*, 120–. ICSM '00, Washington, DC, USA: IEEE Computer Society.

Nam, J., D. Schuler, and A. Zeller. 2011. Calibrated mutation testing. In *2011 ieee fourth international conference on software testing, verification and validation workshops*, 376–381. Washington, DC, USA: IEEE Computer Society.

Offutt, A Jefferson, and Stephen D Lee. 1991. How strong is weak mutation? In *Proceedings of the symposium on testing, analysis, and verification*, 200–213.

Offutt, A. Jefferson, and Jie Pan. 1996. Detecting equivalent mutants and the feasible path problem. In *Proceedings of the 1996 annual conference on computer assurance*, 224–236. Gaithersburg, Maryland: IEEE Computer Society.

Padioleau, Yoann, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. 2006. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *Plos 2006: Linguistic support for modern operating systems*. San Jose, CA: ACM.

Palix, Nicolas, Julia Lawall, and Gilles Muller. 2010. Tracking code patterns over multiple software versions with Herodotos. In *Proceedings of the 9th international conference on aspect-oriented software development*, 169–180. Rennes and Saint Malo, France: ACM.

Palomba, Fabio, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2016. Automatic test case generation: What if test code quality matters? In *Proceedings of the 25th international symposium on software testing and analysis*, 130–141.

Panichella, Annibale, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44(2):122–158.

Petrovic, Goran, Marko Ivankovic, Bob Kurtz, Paul Ammann, and Rene Just. 2018. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 ieee international conference on software testing, verification and validation workshops (icstw)*, 47–53. IEEE.

Robinson, Peter N, Rosario M Piro, and Marten Jäger. 2017. Jannovar. In *Computational exome and genome analysis*, 203–208. Chapman and Hall/CRC.

Saleh, Iman, and Khaled Nagi. 2015. Hadoopmutator: A cloud-based mutation testing framework. In *International conference on software reuse*, 172–187. Springer.

Sanfilippo, Salvatore. 2017. Redis.

Śliwerski, Jacek, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes* 30(4):1–5.

Tomassi, David A., Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. To appear in Proceedings of the 41st ACM/IEEE International Conference on Software Engineering ICSE 2019, Montreal, Canada, May 25 - 31, 2019.

Torvals, Linus. 2017. Linux kernel.

Vokolos, Filippos I., and Phyllis G. Frankl. 1998. Empirical evaluation of the textual differencing regression testing technique. In *1998 international conference on software maintenance, ICSM 1998, bethesda, maryland, usa, november 16-19, 1998*, 44–53. Washington, DC, USA: IEEE Computer Society.

Wang, Zi, Ben Liblit, and Thomas Reps. 2020. Tofu:target-orienter fuzzer. 2004.14375.

Wong, W. Eric, Joseph Robert Horgan, Aditya P. Mathur, and Alberto Pasquini. 1999. Test set size minimization and fault detection effectiveness: A case study in a space application. *Journal of Systems and Software* 48(2):79–89.

Wu, Zhengkai, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 2019. Reinam: Reinforcement learning for input-grammar inference. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 488–498. ESEC/FSE 2019, New York, NY, USA: Association for Computing Machinery.

Xin, Qi, and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Issta*, vol. 17, 226–236.

Yang, Qian, J Jenny Li, and David M Weiss. 2009. A survey of coverage-based testing tools. *The Computer Journal* 52(5):589–597.

Zalewski, M. 2020. American fuzzy lop.