Automatic Program Repair Using Genetic Programming

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Claire Le Goues

May 2013

Abstract

Software quality is an urgent problem. There are so many bugs in industrial program source code that mature software projects are known to ship with both known and unknown bugs [1], and the number of outstanding defects typically exceeds the resources available to address them [2]. This has become a pressing economic problem whose costs in the United States can be measured in the billions of dollars annually [3].

A dominant reason that software defects are so expensive is that fixing them remains a manual process. The process of identifying, triaging, reproducing, and localizing a particular bug, coupled with the task of understanding the underlying error, identifying a set of code changes that address it correctly, and then verifying those changes, costs both time [4] and money, and the cost of repairing a defect can increase by orders of magnitude as development progresses [5]. As a result, many defects, including critical security defects [6], remain unaddressed for long periods of time [7]. Moreover, humans are error-prone, and many human fixes are imperfect, in that they are either incorrect or lead to crashes, hangs, corruption, or security problems [8]. As a result, defect repair has become a major component of software maintenance, which in turn consumes up to 90% of the total lifecycle cost of a given piece of software [9].

Although considerable research attention has been paid to supporting various aspects of the manual debugging process [10, 11], and also to preempting or dynamically addressing particular *classes* of vulnerabilities, such as buffer overruns [12, 13], there exist virtually no previous automated solutions that address the synthesis of patches for general bugs as they are reported in real-world software.

The primary contribution of this dissertation is GenProg, one of the very first automatic solutions designed to help alleviate the manual bug repair burden by automatically and generically patching bugs in deployed and legacy software. GenProg uses a novel genetic programming algorithm, guided by test cases and domain-specific operators, to affect scalable, expressive, and high quality automated repair. We present experimental evidence to substantiate our claims that GenProg can repair multiple types of bugs in multiple types of programs, and that it can repair a large proportion of the bugs that human developers address in practice (that it is *expressive*); that it scales to real-world system sizes (that it is *scalable*); and that it produces repairs that are of sufficiently *high quality*. Over the course of this evaluation, we contribute new benchmark sets of real bugs in real open-source software and novel experimental frameworks for quantitatively evaluating an automated repair technique. We also contribute a novel characterization of

the automated repair search space, and provide analysis both of that space and of the performance and scaling behavior of our technique.

General automated software repair was unheard of in 2009. In 2013, it has its own multi-paper sessions in top tier software engineering conferences. The research area shows no signs of slowing down. This dissertation's description of GenProg provides a detailed report on the state of the art for early automated software repair efforts.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Claire Le Goues

This dissertation has been read and approved by the Examining Committee:

Westley R. Weimer, Advisor

Jack W. Davidson, Committee Chair

Tai Melcher

Stephanie Forrest

Anita Jones

Accepted for the School of Engineering and Applied Science:

James H. Aylor, Dean, School of Engineering and Applied Science

May 2013

"One never notices what has been done; one can only see what remains to be done."
–Marie Curie

Acknowledgments

"Scientists have calculated that the chances of something so patently absurd actually existing are millions to one. But magicians have calculated that million-to-one chances crop up nine times out of ten."

– Terry Pratchett

It is impossible for me to overstate my gratitude to Wes Weimer for teaching me everything I know about science, despite what has been charitably described as my somewhat headstrong personality. He encourages the best in me by always expecting slightly more, and I hope that he finds these results adequate. He has also been an incomparable friend.

Most graduate students are lucky to find one good adviser; I have had the tremendous fortune to find two. Stephanie Forrest taught me any number of things that Wes could not, such as the value of a simple sentence in place of a complicated one. I am very thankful for her friendship and mentorship.

I thank the members of my family for their love, guidance, support, and good examples over all 28 years, not just these six. They manage to believe in my brilliance unconditionally and rather more than I deserve without ever letting me get ahead of myself, which is a difficult balance to strike.

I have been blessed with a number of truly excellent friends who support me, challenge me, teach me, and make me laugh on a regular basis, both in and out of the office. They show me kindness beyond what is rational, from letting me live in their homes for indeterminant periods of time to reading this document over for typos and flow, simply because I asked. I hope they know how much it has meant to me. I am equally grateful to the Dames, Crash in particular, for teaching me to be Dangerous.

Finally, I must acknowledge and thank my brilliant and loving Adam. His love is unfailing and his support ranges from the emotional to the logistical to the distinctly practical (e.g., doing of dishes, feeding of pets). I am regularly flummoxed by how lucky I am to have him in my life.

Contents

List of Tables

List of Figures

List of Terms

Note: terms are linked to their associated glossary entry at their first introduction or definition, and again at their first usage within each subsequent chapter. Within this glossary, terms are linked more comprehensively.

abstract syntax tree — the tree representation of the syntactic structure of source code. Each node of the tree corresponds to a programming language construct in the code [14]. The tree is *abstract* in that it does not represent every concrete detail in the code, e.g., parentheses are omitted because they are implied by the grammatical structure of the programming language. Abstract syntax trees are related to the program's parse tree, can be efficiently constructed, and can losslessly represent all structured programs. ix, 32, 34, 88

Amazon Web Services — a popular provider of online cloud computing services, such as compute time and virtual machine instances at hourly rates, and data storage. ix, 12, 75, 101

bug — a colloquialism referring to a mistake in a program's source code that leads to undesired behavior when the program is executed, whether the undesired behavior is a deviation from the functional specification, an explicit security vulnerability that may be maliciously exploited, or a service failure of any kind. ix, x, 1, 10, 24

bug report — a collection of pre-defined fields, free-form text, and attachments, to aid in the triage, management, and repair of a bug, typically in the context of a bug-tracking database. ix, x, 9

bug-tracking database — a system for reporting and managing problems, bugs, and requests for a software system. Such databases store bug reports, and in many cases accept submissions from both developers and users of a given system . ix, 2, 14, 94

chromosome — in the context of a genetic algorithm, synonym of individual. ix

cloud computing — a broad term referring to computing resources (such as compute time or data storage) delivered as a network service, especially over the internet. ix, 7, 12, 35, 74, 101

code bloat — in the context of a genetic programming algorithm, the tendency of such algorithms to produce unnecessary intermediate code (i.e. introns) that does not contribute to the utility of the eventual solution [15].. ix, x, xii, 25

correct service — as delivered by a software system that performs its intended function. ix, 10

crossover — in the context of a genetic algorithm, the mechanism by which two existing individuals (parents) exchange portions of their genome to produce new individuals (offspring). ix, xi, xiii, 20, 25, 29, 88

debugging — the process of learning the details of a program's specification and implementation to understand a bug corresponding to a deviation between the two, and then identifying and validating a set of changes to the source code or configuration that addresses that deviation. ix, 17

defect — See bug. ix, 1, 24, 112

defect report — see bug report. ix, xiii, 2, 14, 94

error — in the context of software behavior, either a deviation between a system's implementation and its specification or its specification and requirements that leads to a service failure that can manifest, colloquially, as a bug or defect. ix, x, 1, 10

execution-based testing — the process of gaining confidence that a software system is meeting its specification by running it on a set of inputs and comparing the output on a set of corresponding expected outputs. ix, 11, 61, 113

expressive — We consider a technique for automatic program repair expressive when it can repair a variety of types of bugs in a variety of types of programs, and when it can repair a non-trivial proportion of bugs that humans developers address in practice. ix, xi, 3, 24, 46, 71, 87, 113

fault — in the context of software behavior, the hypothesized or adjudged cause of an error. ix, 10

fault localization — a technique for automatically identifying smaller sets of lines of source code likely to be implicated in a particular bug or error, with the goal of reducing the amount of a program's source code that must be examined to identify a reasonable fix. ix, 3, 17, 25, 88

final repair — in the context of GenProg, the result of minimizing the initial repair to remove changes that are unnecessary to effect the desired change in program behavior. Such changes may arise because of the tendency of genetic programming algorithms to produce extraneous code, referred to as the code bloat problem. ix, xii, 30, 44, 51

fitness — in the context of a genetic algorithm, a quantitative measure of an individual's proximity to the solution. Measured by the objective function or fitness function. ix, xi, 4, 20

fitness function — the objective function in a genetic programming search, measuring the desirability of a candidate solution. ix, xi, 4, 20, 25, 87

functional requirement — the output that a software system is intended to compute based on its input. As distinguished from a non-functional requirement. ix, xii, 1, 10, 29, 60

fuzz testing — a technique that automatically generates fault-identifying test suites by generating random inputs. ix, 12, 47

generation — in the context of a genetic algorithm, corresponds to one iteration of the algorithm, including the selection of individuals from the previous generation's population, the mutation and recombination (crossover) of those individuals to produce offspring individuals, and the evaluation of the fitness or desirability of the individuals in the population. ix, xii, xiii, 20, 30, 32

genetic algorithm — a stochastic search and optimization strategy that mimics the process of biological evolution. ix–xiii, 4, 13, 19

genetic programming — the application of genetic algorithms to source code, seeking to discover computer programs tailored to a particular task. ix, x, xiii, 4, 13, 20, 25, 87, 113

high quality repair — We consider a repair to be of high quality if it fixes the underlying cause of a defect (i.e., avoids overfitting), maintains the program's other required functionality, and avoids the introduction of new vulnerabilities or defects. ix, xi, 3, 24, 46, 71, 87

human competitive — We consider a technique for automatic program repair human competitive when it is scalable, when it can apply to a wide variety of bugs in a variety of program types (that is, it is expressive), when it generates high quality repairs, and when it can repair such bugs at a time and monetary cost that is comparable to those expended on human repair efforts. ix, xiii, 2, 24, 47, 87, 113

individual — in the context of a genetic algorithm, corresponds to a candidate solution to the optimization problem the algorithm is attempting to solve. An individual is encoded as a set of properties (conceptually, its genes) that can be mutated, recombined, or altered. Also referred to as a *chromosome* or *variant*. ix, xi–xiii

initial repair — in the context of GenProg, the first solution patch identified on a given run of the algorithm that, when applied to the input program, results in a program that passes all of the test cases, including the negative test cases

and the positive test cases. The initial repair may contain extraneous edits that are not necessary to affect the change in behavior, a result of the code bloat problem. The initial repair may be minimized to result in a smaller final repair. ix, x, 30, 44, 51

local optimum — a local maximum of a function, that is, maximal as compared to the immediate neighbors of the given point, but not necessarily maximal as defined by the global maximum of the function. ix, 19, 29

mutation — in the context of a genetic algorithm, an operator that generates a new candidate solution by modifying one gene or component of an existing individual. ix, xi, xiii, 20, 25, 29, 88

negative test case — in the GenProg algorithm, an initially failing test case that encodes the buggy behavior to be repaired automatically. The goal of the repair search is to find a patch that causes such test cases to pass while maintaining the behavior encoded by the positive test cases. ix, xi, xii, 29, 50, 76, 90

non-functional requirement — a quality-of-service expectation regarding the computation of a software system, in terms, e.g., of security, time, or memory requirements for a computation. As distinguished from a functional requirement. ix, xi, 10, 29, 60

objective function — a function to be optimized in the context of an optimization or search problem. In our application, also referred to as the fitness function. ix, xi, 4, 20, 37

oracle comparator — in the context of software testing, a mechanism that produces an expected result for a test case combined with a mechanism that checks the actual against the expected result. ix, 3, 11

patch — a set of changes to program source or binary, often though not exclusively to fix a bug. In this dissertation, a patch operates at the source-code level. ix, xii, 2, 24, 46

population — in the context of a genetic algorithm, the collection of individuals corresponding to one generation of candidate solutions. ix, xi, 25, 30

positive test case — in the GenProg algorithm, an initially passing test case that encodes correct behavior that the program should maintain post-repair. The goal of the repair search is to find a patch that causes the negative test cases to pass while maintaining the behavior encoded by the positive test cases. ix, xii, 29, 51, 76, 90

repair — a patch intended to fix a bug or vulnerability. ix, xi, 4, 17, 24

scalable — also referred to as a system's scalability. We consider an automatic program repair solution scalable when it can repair bugs in large, real-world legacy systems comprising thousands to millions of lines of code at a feasible (or human competitive) time and monetary cost. ix, xi, 3, 24, 71, 87, 113

security vulnerability — a software error that allows a malicious attacker to reduce or subvert the service provided by a system. ix, 1, 10, 24, 46

selection — in the context of a genetic algorithm, the process by which individuals in the current generation are chosen for continued mutation and recombination (crossover). ix, 4, 25, 30

service failure — in the context of software behavior, the state in which the service delivered by a piece of software deviates from correct service, as defined by its specification or requirements. ix, x, 1, 10

stochastic search — optimization algorithms that incorporate probabilistic elements, such as simulated annealing, hill climbing, random search, and genetic programming. ix, xi, 3, 13, 24, 71, 113

test case — following Binder *et al.* [16], the combination of a program input, the expected output of the program on that input, and an oracle comparator that can validate whether the observed output of the program on the input matches the expected output. The input may be human-written, taken from a regression test suite, adapted from a bug report that includes steps to reproduce an error, or generated automatically. ix, xii, xiii, 3, 11, 24, 28, 113

test suite — a collection of test cases. ix, xiii, 5, 11

test suite prioritization — a technique that seeks to reduce the cost of software testing by identifying orderings for the execution of a test suite that increase the probability of identifying faults or to cover as much of the code as possible as early in a run of the test suite as possible. ix, 13, 92

test suite reduction — a techniques that seeks to reduce the cost of software testing by identifying a maximally informative subset of an existing test suite; the subset should be less expensive to run than the full set, but still be of high quality (e.g., display high coverage or fault-identification probability). ix, 13

triage — in the context of software engineering, and specifically bug reporting, the process of reviewing a defect report to determine whether it contains sufficient and suitable information to be addressed and, if so, to assign the report to a developer or team to address it. ix, 2, 14

variant — in the context of a genetic algorithm, synonym of individual. ix, 4

version control system — also known as revision control or source control, a system that assists in the management of changes to program source and other documents. Changes are usually identified by a number termed the

"revision." Each revision is associated with a time-stamp and the person making the change and a commit message. An instance of a version control system is typically called a "repository." Common systems include CVS, SVN, Mercurial, and Git. ix, 2, 72, 94

Chapter 1

Introduction

"Everyday, almost 300 bugs appear [. . .] far too many for only the Mozilla programmers to handle"

– Mozilla Developer, 2006 [17, p. 363]

SOFTWARE quality is an urgent problem, and bugs that cause unintended program behavior are prevalent in modern software systems. The International Software Testing Qualifications Board defines software errors, defects, and failures, including an informal definition of "bug", as follows:

> A human being can make an error (mistake), which produces a *defect* (*fault*, *bug*) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure. Defects in software, systems or documents may result in failures, but not all defects do so [18, emphasis added].

In this dissertation, we use the terms bug and defect interchangeably to refer to the colloquial meaning, that is, mistakenly-formed program source code that leads to undesired behavior when the program is executed, whether the undesired behavior is a deviation from the functional specification, an explicit security vulnerability that may be maliciously exploited, or a service failure of any kind [19].

By any definition, mature software projects ship with both known and unknown bugs [1], and the number of outstanding defects typically exceeds the resources available to address them [2]. Software defects are expensive, both in terms of lost productivity caused by faulty software and in terms of the cost of paying developers to fix bugs in the software that companies sell. A 2008 FBI survey of over 500 large firms reported that security defects alone cost such firms $289,000 annually [20, p.16]; a 2002 NIST survey calculated that overall, software errors in the US cost 59.5 billion (0.6% of GDP) annually [3]. A 2008 survey of 139 North American firms found that each spent an average of $22 million annually fixing software defects [21].

In other words, software defects are prevalent and problematic, and fixing them is a dominant practical concern. Software repair is a major component of software maintenance, which consumes a up to 90% of the total cost of software production [9]. Much of this cost stems from the fact that the human developer remains the state of the art in bug-fixing practice. Many projects (both open- and closed-source) use tools such as bug-tracking databases and version control systems to manage reported defects and bug-repair efforts [22]. Bug repair efforts in practice thus share common workflow features across developers, projects, and companies. In common practice, when a defect is encountered, a new defect report is written to describe it. Once filed, the report is then triaged to determine if it is valid and unique and, if so, assign it to the appropriate team or developer [7]. Correctly fixing a problem requires a human developer to understand first, what the system implementation is currently doing; second, what the explicit or implicit specification suggests the implementation should be doing instead; third, why they are different; and fourth, what changes to the program source code or configuration will correct the mismatch. These changes correspond to a proposed fix or patch for the bug. In many organizations, that fix will be validated (e.g., through a code review) before being committed or deployed to the users [23].

Although humans programmers are versatile in the types of problems they can address, this process costs time [4] and money, and the cost of repairing a defect can increase by orders of magnitude as development progresses [5]. As a result, many defects, including critical security defects [6], remain unaddressed for long periods of time [7]. Moreover, humans are error prone, and many human fixes are imperfect: in a survey of 2,000 fixes to systems software, Yin et al. found that 14–24% are incorrect and 43% of those bad fixes led to crashes, hangs, corruption, or security problems [8].

This situation motivates a need for automatic solutions to help alleviate the bug repair burden on human developers by *automatically and generically repairing bugs in deployed and legacy software.* To be useful, such a solution should emulate many of the benefits of human programmers, whose ingenuity allows them to repair many different types of bugs in different types of systems, and such a system should be human competitive in terms of expressive power and both time and monetary cost. We therefore seek an automated repair solution that displays several properties:

- **Scalability.** A technique for automated repair should cheaply scale to the large, realistic software systems that exist in practice.

- **Quality repairs.** A technique for automated repair should produce quality bug repairs that address the root of the problem and that do not introduce new crashes or vulnerabilities.

- **Expressive power.** A technique for automated repair should be able to fix a variety and a large proportion of the defects that appear in practice.

Software quality as a concern has motivated considerable previous research. Such previous work serves to help developers construct new systems with fewer bugs; recover from or prevent particular classes of bugs in new or existing

systems; or identify, localize, or understand existing bugs in order to more easily or cheaply repair them. These lines of inquiry have resulted in techniques for building resilient or fault-tolerant systems (e.g., [24, 25]), dynamically detecting and recovering from particular types of faults on-the-fly (e.g., [13]), or tools or procedures for constructing systems with fewer bugs by design (e.g., [26]). There has similarly been significant research in defect detection, prevention, localization, and explanation [1, 27, 28, 29, 30, 31, 32], all with the intention of helping programmers understand and then repair bugs in software.

While this previous work serves a variety of useful purposes, it all, in one way or another, fails to possess one or more of the desired properties of a viable automatic repair solution. Correct-by-construction programming paradigms require the construction of new systems, and do not address the billions of lines of legacy code currently in deployment [33]. Verification and provably-correct programming and repair systems typically do not scale to real-world systems' sizes [34]. Resilient and fault-tolerant system techniques restrict attention to a particular fault class or defect type (e.g., buffer overflows [12] or infinite loops [35]), and are thus insufficiently expressive; and fault localization, detection, prevention, and related work serve to help improve the debugging experience, but do not propose candidate source patches on their own. Currently, a valid bug report in a real-world system must still be patched by hand by a human, which remains a time-consuming and expensive process.

Put simply: software defects are ubiquitous and economically detrimental; finding and repairing them is a difficult, time consuming process that, despite significant research advances, remains manual; and their number and scale preclude their timely and cost-effective manual repair. This situation motivates an automatic solution to alleviate at least some portion of the bug repair burden; this dissertation concerns itself with one such solution.

1.1 GenProg: Automatic program repair using genetic programming

This dissertation presents and evaluates *GenProg*, an algorithm that combines lightweight program analysis techniques like test-case-based fault localization with stochastic search algorithms to automatically, generically, and efficiently repair bugs in real-world, off-the-shelf programs. The algorithm is designed to be human competitive, in that it is scalable, expressive, and cost-effective; and to produce high quality repairs for a broad range of bug and program types. The goal is not to repair *all* defects of a particular type, but rather to repair many defects of many different types, in a variety of programs, with as little *a priori* knowledge about bug or program type as possible.

As input, GenProg takes the source code of an existing program that contains a bug and a set of test cases. We use the term test case to signify the combination of an input to a program, the expected output of the program on that input, and an oracle comparator that can validate whether the observed output of the program matches the expected output for the given input [16]. The input test cases may be human-written; taken from a regression test suite, adapted from a bug report that includes steps to reproduce an error, or generated automatically. The buggy input program fails

at least one of the input test cases: this observes the failure resulting from the defect. When successful, the output of the tool is a set of source code changes—a patch—that, when applied to the initial program, results in a modified program that passes all of the test cases, including those that originally encode the defect. Such a patch therefore retains required functionality while addressing the defect in question. We call such a patch a repair for the bug, and use the terms "repair" and "patch" interchangeably throughout this dissertation.

Our solution relies on three broad algorithmic insights, each of which contributes partially to realizing our goal of a scalable, expressive, high-quality automatic repair solution.

The space of software patches is amenable to exploration via stochastic search. Stochastic or randomized algorithms (e.g., simulated annealing, iterative hill climbing, ant colony optimization, genetic algorithms, etc.) can be well-suited to searching through very large, high-dimensional landscapes of alternatives [36], even when finding optima for imprecise or noisy signals. Such search techniques have been able to rapidly solve a number of interesting and difficult problems (e.g., evolutionary algorithms have produced human competitive results [37] in music synthesis [38], electron microscopy [39] and game solvers [40]).

GenProg's search algorithm is based on genetic programming (GP), a search method inspired by biological evolution that discovers computer programs tailored to a particular task [41, 42]. In our application, the search traverses the space of candidate repairs for a given bug. GP uses computational analogs of biological mutation and crossover to generate new patch variations, which we call variants. In a GP, a user-defined objective function (also called a fitness function) evaluates each variant; GenProg uses test cases to evaluate variant fitness, and individuals with high fitness (that pass many test cases) are selected for continued evolution. Although GP has solved an impressive range of problems (e.g., [43]), it has not previously been used either to evolve off-the-shelf legacy software or patch real-world vulnerabilities, despite various proposals directed at automated coevolution of repairs and test suites (e.g., [44]). Previous efforts to apply search algorithms to automatic error repair have been stymied by scalability concerns [15], which we overcome through novel design decisions (see Chapter 3).

Adapting the GP search framework to a new application requires design decisions about representation, mutation operators, and fitness functions that define the search space. It is otherwise a very flexible framework, which allows us to incorporate existing software engineering techniques like fault localization to constrain the space intelligently and enable scalability to real world systems. GenProg's representation and mutation operators are defined very generically, so it can represent a wide variety of source code change combinations and apply to many different types of bugs in many different types of programs. Finally, GenProg uses the input test cases to define the objective function in terms of expected program behavior, which increases confidence that the generated repair is of high quality.

Test cases provide useful insight into program behavior. GenProg uses test cases to encode both the bug under repair and the existing functionality that should be maintained post-patch; the test suite (collection of test cases) therefore serves as a proxy for a functional specification of the program's requirements. This encourages the creation of high-quality repairs, because expected post-patch functionality explicitly guides the search. Moreover, while they provide weaker correctness guarantees than formal proofs, test suites as measures of correctness scale more readily to real-world systems [45]. Test cases are also far more readily available in practice and easier to automatically generate than formal behavioral specifications [46,47]. For example, none of the 22 real-world, open-source C programs (totaling over 6 million lines of C code) used as benchmarks in this dissertation are formally specified. As a result, using test cases to define correctness enables applicability to legacy systems. Finally, program behavior on test inputs can provide considerable feedback about program semantics and programmer-specified expected behavior, and test cases can flexibly encode a broad set of interesting program behaviors. As a result, they enable an expressive repair system, because they can be used to describe many different types of bugs.

Existing program behavior contains the seeds of many repairs. Fundamentally, most programmers program correctly most of the time: incorrect program behavior, such as a forgotten null check, is often correctly implemented elsewhere in the source code of the same program. This insight, sometimes referred to as the "competent programmer hypothesis", also underlies other research techniques (e.g., [48, 49]). We view existing program behavior as an expression of the domain-specific expertise of the original programmers, and seek to leverage that existing behavior in a search for a repair. GenProg therefore reuses existing program code when trying to fix a bug. Looking for candidate repairs within the same program space provides a useful domain-specific limitation on the search space of possible patches (improving scalability) while taking advantage of the information previous programmers provide in the form of the existing code (enabling expressive power and repair quality).

1.2 Evaluation metrics and success criteria

In general, we evaluate GenProg by using it to find patches for real bugs in real-world systems. The dominant guiding principle in our experimental design is to establish that GenProg is human competitive. As such, we select benchmarks, evaluation metrics, and success criteria with a comparison to human developers in mind. In this context, a benchmark scenario consists of a C program, a set of currently passing test cases for that program, and at least one failing test case that exercises a bug in that program. In some cases, these scenarios may also be associated with a human fix for the defect or other related software engineering artifacts, such as a source code repository commit message or a bug database report. We use different benchmark-selection techniques (i.e., we vary the way we decide which bugs to try to

fix) and evaluation metrics to substantiate the various components of our claims that GenProg is human competitive and that it possesses the properties required of a useful automatic repair technique.

1.2.1 Expressive power

We use two approaches to substantiate our claim that GenProg is expressive.

First, we demonstrate that GenProg can fix a variety of different bug types in a variety of different programs. We develop a benchmark set of several different types of bugs, focusing as much as possible on real-world security vulnerabilities. We then show that GenProg can repair them, to provide an existence proof of its expressive power. The vulnerability types include infinite loops, segmentation faults, remote heap buffer overflows to inject code, remote heap buffer overflows to overwrite variables, non-overflow denials of service, local stack buffer overflows, integer overflows, and format string vulnerabilities. The programs include Unix utilities, servers, media players, text processing programs, and games. Formally, we hypothesize:

> **Hypothesis 1:** Without defect- or program-specific information, GenProg can repair at least 5 different defect types and can repair defects in at least least 10 different program types.

Second, we present an approach for systematically producing a large benchmark set of bugs that is indicative of the defects that humans repaired over a period of time in a program's development history. We then measure the proportion of these historical, real-world bugs GenProg can successfully repair to approximate its applicability to the variety of bugs human programmers address in practice. Although we do not target the repair of all bugs, we aim to develop a *practical* technique, and thus hypothesize:

> **Hypothesis 2:** GenProg can repair at least 50% of defects that humans developers fix in practice.

1.2.2 Scalability

To be considered scalable, GenProg should be able to repair bugs in programs of indicative size at a reasonable cost. We therefore focus on benchmark programs taken from open-source repositories, with an aim of repairing bugs in programs that are indicative of the systems that developers work on. We measure program size by the number of lines of source code as well as the number of test cases in the program's regression test suite. We show repair results on programs of a variety of sizes, up to 3.1 million lines of C code and including up to 10,000 test cases, in the largest instance.

We measure the cost of repair in terms of temporal and economic cost. We measure time to repair in terms of wall-clock time as well as number of fitness evaluations to a repair, which is a program, hardware, and test-suite independent measure of repair effort. We measure economic cost in US dollars by running experiments in a purchasable

commodity cloud computing system; more detail, especially regarding the definition of human competitive cost, is given in Chapter 5.

Formally, we hypothesize:

> **Hypothesis 3:** GenProg can repair bugs in programs of up to several million lines of code, and associated with up to several thousand test cases, at a time and economic cost that is human competitive.

We also provide quantitative and qualitative analysis of the size of the search space and the factors that influence GenProg's scalability.

1.2.3 Repair quality

Finally, to qualify as useful, GenProg must produce high quality patches in practice. In this dissertation we say a repair is of high quality when it simultaneously (A) fixes an underlying defect while (B) maintaining other required functionality that the program previously implemented and (C) avoids the introduction of new vulnerabilities or defects.

Repair quality, and in particular how to measure it, is an open and interesting research problem; we note that human developers are imperfect in this respect, and that many human-developed patches are problematic in practice [8]. Other definitions of repair quality (e.g., maintainability [50]) have been considered in the literature but are beyond the scope of this work.

To substantiate our claim that GenProg patches are of sufficient quality, we first present a number of qualitative explanations of patches that GenProg has found. Second, we develop quantitative measures of repair quality by adapting industrial techniques for vetting security-critical patches to our application. We measure patched program performance on held-out workloads, large suites of held-out tests, and exploit variant bug-inducing inputs — in all cases, observing negligible service degradation on the case study programs in question. We hypothesize:

> **Hypothesis 4:** GenProg-produced patches maintain existing program functionality as measured by existing test cases and held-out indicative workloads; they do not introduce new vulnerabilities as measured by black-box fuzz testing techniques; and they address the underlying cause of a vulnerability, and not just its symptom, as measured against exploit variants generated by fuzzing techniques.

1.3 Contributions and outline

The overarching thesis of this dissertation is:

> **Thesis:** stochastic search, guided by existing test cases, can provide a scalable, expressive, and human competitive approach for the automated repair of many types of defects in many types of real-world programs.

The primary contributions of this dissertation are:

- GenProg, a novel algorithm that uses genetic programming to automatically repair bugs in off-the-shelf, legacy C programs (Chapter 3).
- Empirical evidence substantiating the claim that GenProg is expressive (Chapter 4) in that it can repair many different types of bugs in different types of programs.
- Empirical evidence, produced via a novel experimental framework, substantiating the claim that GenProg produces high quality repairs (Section 4.4). We provide a qualitative discussion of a number of examples to illustrate the types of repairs that GenProg finds as well as a quantitative evaluation of GenProg's effect on real software systems.
- Empirical evidence substantiating the claim that GenProg is human competitive in expressive power and cost (Chapter 5).
- The ManyBugs benchmark set, and a system for automatically generating such a benchmark set, for evaluation and analysis of automatic program repair techniques (Section 5.3).
- Analysis of the factors that influence repair success and time-to-repair, including a large-scale study of representation, operators, and the search space of genetic programming for automatic program repair (Chapter 6).

In addition to the chapters corresponding to contributions, Chapter 2 provides background and related work on software engineering. Specifically, it discusses several techniques practitioners and researchers use to build systems without bugs, and to find and fix bugs once those systems are built. It also provides an overview of metaheuristic search in general and genetic algorithms and genetic programming in particular. Chapter 7 concludes.

Chapter 2

Background and Related Work

BUG fixing, software testing, and software quality are dominant concerns in software engineering practice. These concerns motivate the automated repair work presented in this dissertation. This chapter concerns itself with background required to understand that work and place it in context.

A central concern to automated bug repair is the nature of bugs in software. Thus, we begin by defining a bug and how it relates to the intended function of a piece of software (Section 2.1). We discuss ways that developers gain confidence that programs are correct and free of bugs (Section 2.2), specifically with respect to testing techniques (Section 2.2.1). Bugs occur despite these efforts, and we next discuss ways that developers and companies learn about and fix bugs, including the creation and use of bug reports (Section 2.3.1), as well as research and industrial methods for automatically finding bugs in software (Section 2.3.2). This discussion transitions into a discussion of how human developers, who remain the state-of-the-art in bug-fixing practice, repair software defects (Section 2.4). Section 2.4 also discusses ways we might evaluate the quality of a given bug fix.

The process of fixing a bug by hand has many characteristics of a search problem, where the objective is a set of changes to program source code that suitably addresses the problematic behavior. This insight forms the basis of our technique for automatic program repair. This chapter therefore also provides background on metaheuristic search strategies in general (Section 2.5), and genetic algorithms in particular, before concluding with a discussion of previous work in automatic bug preemption and repair (Section 2.6).

2.1 What is a bug?

The area of dependable computing has devoted considerable attention to precisely defining defects, faults, and errors in the context of software and program source code. We provide a brief overview here, but direct the interested reader to,

for example, Avizienis *et al.* [19] (on which much of this discussion is based) or Laprie *et al.* [51] for a more thorough treatment.

The intended behavior of a software system is refined from the requirements that the system is designed to meet, which may include both the answer it is expected to compute as well as qualities related to its evolution and execution (for example, quality of service concerns such as how securely or efficiently that answer is produced). The former conditions are called the system's functional requirements, while the latter are called its non-functional requirements. Both types of properties are important in the construction of systems that meet human requirements of software systems, though in this work we largely restrict our attention to functional requirements; the primary non-functional requirement we consider is execution time. A functional specification refines and describes the overall requirements, both in terms of behavior and performance. Such a specification may take many forms, including a formal model of desired behavior expressed in first-order logic; architecture or modeling diagrams in one of many languages designed for the purpose; prose, such as natural-language documentation; etc.

Regardless of the level of formality, all software systems have an intended function, and correct service is delivered when the system implements this function. When the delivered service deviates from correct service, a service failure, or just *failure*, is said to occur. This may happen either because the implementation of the system does not comply with its specification or because the specification was incomplete or incorrect. This deviation is called an error, and the cause of the error (incorrect code or configuration, for example) is a fault. A fault may be active or dormant, as faulty code that is never executed may never lead to a visible error. The term bug is a colloquialism that refers broadly to a problem in a program's source code or configuration (fault) that leads to incorrect or unexpected behavior (error). We use the terms bug and defect interchangeably in this dissertation to refer to this colloquial meaning.

There are a multitude of ways that programs can fail to fulfill their intended service [52], and bugs arise from both intended (malicious) faults in the code as well as unintended ones. One important class of coding errors is those that lead to security vulnerabilities, or errors that allow a malicious attacker to reduce or subvert the service provided by a system. Such vulnerabilities vary widely in the mechanisms that attackers may use to exploit them, they have been taxonomized in many ways [53, 54]. Common breakdowns of existing vulnerability types include improper input validation (e.g. SQL injection vulnerabilities [55, 56], cross-site scripting attacks [57]), improper synchronization (e.g. timing attacks [58]), improper change management (leading to time-of-check vs. time-of-use errors [59, 60]), and improper allocation and deallocation (e.g., buffer overruns [61, 62], which often lead to illicit control transfer [63]). These types of errors vary considerably in the appropriate approaches to manage them because of the differences in their causes, and have been extensively studied in the research literature [64].

Security-critical bugs are particularly important, but even errors that do not lead to maliciously-exploitable vulnerabilities impact productivity and market share. Infinite loops have been shown to cause significant unresponsiveness in commonly used applications [65, 66]. Such defects negatively impact both market share and reputation of the companies

that sell the software in question, as well as the users of the software, measurably impacting the economy [3]. There is therefore strong impetus to avoid introducing any type of bug during development; we discuss methods for doing so in the next section.

2.2 How do developers avoid bugs?

Ideally, software engineers would write code that is free of errors in the first place. In this section, we review background on strategies that aim to increase the correctness of code as it is written, evolved, and deployed.

2.2.1 What is testing?

Software testing is one of the most important ways that developers gain confidence that a program implementation meets is specification as refined from its requirements. Testing is both a well-established and an active research area [67]. For a more thorough discussion, see for example Amman and Offut [68].

One broad class of software testing activities takes the form of running the program on inputs for which the correct behaviors are known in advance, and comparing the expected output to reality. Formally, this may involve executing a program on a set of inputs and then using a comparator to determine if the result of the execution (the observed output) matches the expected output [16]. This set of activities is referred to as execution-based testing [69] and is the focus of the testing referred to in this work. A test case is defined as an input-output pair and an oracle comparator that is used to determine if the observed output of the tested program on the input matches the expected output. A test suite is a collection of test cases.

Testing activities fall into several categories, including unit and integration testing, which test modules individually and when interacting, respectively, typically during development; regression testing, which checks that code changes do not introduce new or reintroduce old defects; and alpha and beta testing, which tests the code in increasingly realistic operational contexts. Many programs are developed with test suites that fall into some or all of these categories, especially unit and regression tests.

The development and deployment of adequate test suites remains an unsolved problem. Test suites and testing frameworks are often inadequate [3], in that they do not sufficiently exercise functionality to identify faults or determine that a program does everything it is supposed to do. However, even the definition of adequacy as it applies to test suites has been a subject of debate, and researchers have developed two major complementary types of metrics for evaluating test suite quality:

- **Code coverage** approximates what portion of the code is exercised by a test suite, and is measured by mapping each test case to the pieces of code it exercises dynamically. This mapping varies by granularity, which includes

statement, branch, function, and decision levels. Finer-granularity coverage is typically more expensive to compute. The goal is typically to maximize coverage, based on the assumption that a test suite that executes as much code as possible has an increased likelihood of identifying defects.

- **Fault detection power** measures a test suite's ability to identify faults in the code. Techniques that measure fault detection ability introduce faults [16] into the code through manual injection or mutation [70] and then measure how many faults the test suite uncovers.

Testing can be expensive, first because writing appropriate and comprehensive test suites is time-consuming, and maintaining them as the system evolves represents an ongoing cost. Moreover, testing involves running a program repeatedly, which can be very time-intensive, and regression test suites that only run overnight or on weekends are common in industrial practice [71].

2.2.2 How can testing be improved?

Researchers and industrial practitioners have developed, and continue to develop, many approaches to improve testing [72]. We focus on background work dedicated to both increasing quality of test suites and decreasing the costs of writing, running and maintaining them.

Automatic test suite *generation* seeks to generate high-quality test suites. One well-established technique for generating fault-identifying test suites is fuzz testing, which generates test inputs randomly [73]. This idea has been expanded considerably, and test suite generation has been done by analyzing source code exclusively [74, 75, 76], executing the source code repeatedly to learn about various execution paths [77, 78] and using a combination of both approaches [79, 80, 81, 82, 83]. The goals of these approaches range from generating test suites with high coverage or fault prediction capabilities to quickly generating fault-eliciting inputs or signatures [84, 85, 86], to augmenting existing test suites to achieve requirements coverage [87]. A limitation of such techniques is that they typically generate only *inputs*, and not the associated outputs and oracle comparators, though recent development in this area is encouraging [47, 88].

Test suites remain expensive to run, however, in both temporal and economic terms. Industrial practitioners have begun to reduce these costs deploying testing frameworks in cloud computing settings. *Cloud computing* refers to computing resources, such as storage or CPU hours, delivered as a network service. Publicly-available cloud computing resources that allow users to purchase compute time at hourly rates (such as those provided by Amazon Web Services, or AWS) now serve as a backend for many popular internet services (such as Netflix). Companies such as Google often have their own internal cloud computing resources that are used for development, engineering, and testing purposes [89]. Cloud computing resources have also begun to gain traction in research applications [90] (including our own), as they are flexible, broadly available, and provide a grounded mechanism for measuring the economic costs of research

techniques (researchers pay for the experiments, and report on the cost of using their approach to achieve various results).

Researchers have proposed prioritization (e.g., [91]) and reduction (e.g., [92]) techniques to deal with the high cost of testing by identifying informative subsets or orderings of comprehensive test suites. Prioritization produces a permutation of a test suite that is more likely to find faults more quickly than the original. Reduction removes test cases from a test suite to minimize execution time while still covering as many requirements as possible. *Impact analysis* [93, 94, 95] analyzes both the program and the associated test suites to determine which tests might possibly be affected by a source code change, which in turn also reduces testing costs by identifying the subset of a full test suite that should be run to check the effect of a particular source code modification.

Researchers in Search-Based Software Engineering (SBSE) (see Harman [96] for a detailed survey) use evolutionary and related stochastic search methods to improve testing, especially in terms of runtime e.g., to develop test suites, reduce their run time, or improve their quality [97, 98]. Stochastic search algorithms, such as simulated annealing, hill climbing, random search, and genetic programming, are optimization algorithms that incorporate probabilistic elements. Of particular interest in this dissertation are genetic algorithms (GAs) and in particular genetic programming (GP). A genetic algorithm is a heuristic search technique that mimics the process of biological evolution; genetic programming applies genetic algorithms to program source code [41, 42]. We give background on both techniques in Section 2.5. SBSE also uses evolutionary methods to improve software project management and effort estimation [99], find safety violations [100], and in some cases re-factor or re-engineer large software bases [101].

2.2.3 What are formal methods for writing bug-free software?

Testing can be contrasted with verification [102, 103], which proves the correctness of an implementation with respect to a formal specification or property. The guarantees provided by verification are typically stronger than those provided by testing, as they rely on formal mathematical proofs rather than on empirical evidence of correct behavior. However, verification techniques require formal correctness specifications, which are much less common in industrial practice than test cases. They also typically require heavyweight formal tools such as theorem provers [45, 104] or proof assistants [105, 106] and thus do not often scale to real-world systems [34], because they are both time-intensive and difficult to use. Considerable advances have been made in the efficiency of theorem provers, however, and verification approaches have seen considerable improvements in applicability and scalability over time [102].

Program synthesis goes a step further than verification by attempting to discover or construct a provably correct program from a set of constraints that encode user intent [26]. Full *deductive synthesis*, which generates programs through iterative refinement of the specification using well-defined proof rules is a long-established research goal [107, 108]. While there have been successful systems to arise from this line of work [109, 110], such approaches do not

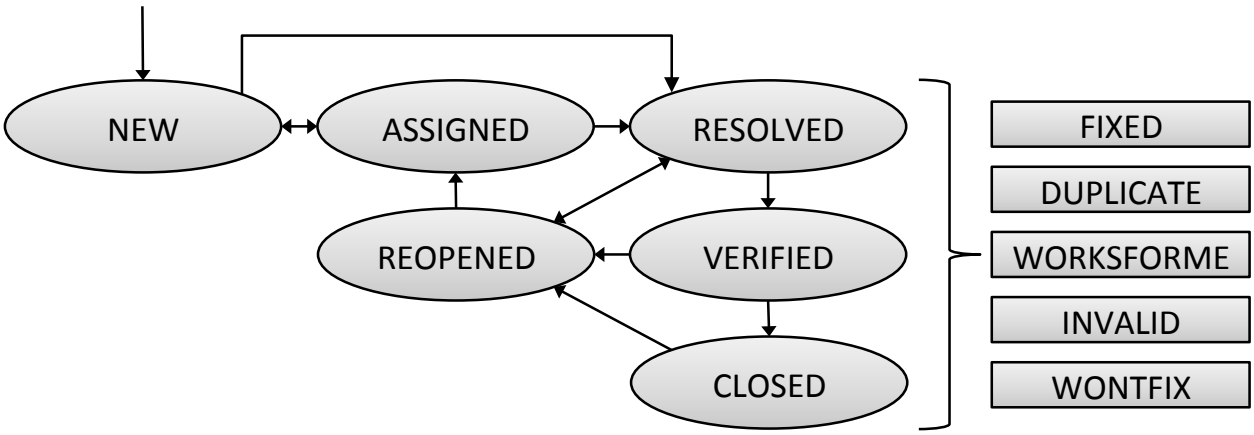

Figure 2.1: The life-cycle of a bug report in the Eclipse open-source project; figure adapted from Anvik *et al.* [17]

provide considerable automation because they require substantial programmer expertise, and have thus historically only gained traction in safety-critical applications, for which such efforts are justified [111]. *Inductive synthesis* generalizes from instances or examples to create programs that provably meet specifications. This type of approach includes those that synthesize from partial, buggy, or incomplete programs [112, 113], which could potentially be extended to fix existing buggy implementations. In general, these techniques have been most successful when restricted in their domain [114]. They otherwise face the same scalability concerns that arise in formal verification. See Gulwani [26] for a more comprehensive overview of program synthesis, the problem area, the search space, and existing approaches.

2.3 How are bugs identified and reported?

Despite testing and verification efforts, bugs may be found in any lifecycle phase of a software project, from the earliest of development to well past deployment [5]. A bug in source code may be identified by its original developer, a testing or quality assurance engineer, an automated bug-finding tool, or by any user who observes unexpected behavior. In this section, we discuss the ways bugs are found as well as how they are typically reported and tracked.

2.3.1 How are bugs reported and managed?

Large projects, both open-source and proprietary, often use tools such as bug-tracking databases to manage bugs and their repairs [22]. A bug-tracking database documents bugs with defect reports (also referred to as *bug reports*) that help developers identify, triage, and, if applicable, reproduce, understand, and ultimately repair a bug. A defect report is a collection of pre-defined fields, free-form text, and attachments. The pre-defined fields include categorical data that may or may not change over time (e.g., the developer to whom the report is assigned or its declared severity may change; the date on which it was reported may not). The free-form text often includes a description of the bug or steps

to reproduce it. Reports are typically associated with developer discussions about possible fixes or related errors or information. Attachments may also be provided, such as a screen capture displaying the observed defective behavior. The database systems usually track dependencies between reports as well as an activity log for each report, such as when a developer is assigned to the report or when its priority has changed [17].

As an example, Figure 2.1 diagrams the life-cycle of a bug report in the issue-tracking system for the Eclipse open-source project. When a new bug report arrives in a system, it is typically first triaged to determine if it is unique and valid and if it provides sufficient information to aid developers in understanding the underlying issue. If it is deemed both valid and unique, it can be assigned to a developer or team for review. The developer may evaluate the bug in question (potentially attempting to reproduce it) at which point some such bugs may be resolved as invalid, unimportant, or not reproducable. Otherwise, the developer may then start to develop a set of changes (a candidate patch) to apply to the system source code or configuration to repair it. Once produced, a candidate patch is often subject to a number of checks to verify its correctness, such as impact analysis, regression testing, or an external review process (e.g., using a code review). If these checks are successful, the patch can be applied and the bug report can be closed.

The arrows between the stages in Figure 2.1 do not signify 100% flow of reports, and a report may transition through a given phase multiple times. For example, a bug report may be spuriously closed as "works-for-me" only to be reopened if another developer successfully reproduces the issue. Additionally, a significant fraction of submitted bug reports are duplicates of those already in the system; previous work suggest that 25–35% of submitted bug reports are in fact duplicates [17, 115]. As a result, not all bug reports ultimately correspond or lead to a patch to the source code. However, to be considered "resolved," all submitted bug reports must be reviewed to determine if they are, in fact, duplicates or otherwise invalid, or whether they correspond to an issue that a developer should address. Triage, duplicate detection, and actual bug fixing are all required for successful maintenance.

Defect reports may be provided by external users of a system who observe and then manually report defective behavior, developers who identify bugs through the course of development, program analysis tools that seek to identify candidate bugs (see Section 2.3.2), or by crash reporting tools that automatically compile reports based on defective behavior experienced by users (these may also ask for input from the user to augment the report). A well-known example of the latter is Dr. Watson, the automatic crash reporting tool associated with Microsoft Windows [116]. This type of tool can provide useful field data to developers. However, bugs that manifest post-deployment are expensive to fix [5] and cost companies in terms of both market share and reputation [66]. Additionally, post-deployment crash reporting can compromise sensitive user data, such as credit cards or passwords.[1] These factors provide a strong motivation for finding and fixing bugs well before they affect software users.

[1] http://windows.microsoft.com/en-US/Windows/microsoft-error-reporting-privacy-statement — "Reports might unintentionally contain personal information . . . your name, part of a document you were working on, or data that you recently submitted to a website."

2.3.2 How can bugs be found automatically?

Because of the high cost of post-deployment defects that affect users, considerable attention has been devoted to identifying bugs in source code during development. Numerous tools have thus been developed over at least 40 years to analyze programs and identify potentially faulty code. The general problem of automatically correctly detecting all bugs in programs is undecidable [117]. As a result, bug-finding tools may produce false alarm bug reports (false positives), may fail to report actual bugs (false negatives), or both. A technique that provably avoids either false positives or false negatives within the context of the technique is called sound; other best-effort techniques are called unsound.

Tools that find candidate bugs in source code have three basic components:

- **A definition of correct behavior.** Automatic bug-finding techniques rely on either explicit (user- or programmer-provided) or implicit or well-established external specifications of correct behavior, such as "should not crash" [84] or "should not contain race conditions" [118]. Specification mining tools [119] learn program-specific rules for correct behavior by mining from existing behavior [120] or source code [121]; learned rules may then be used to identify potentially faulty code. The most successful of bug-finding tools (in terms of scalability and commercial viability) rely on a mix of implicit specifications and human-defined patterns of potentially buggy behavior [30, 33].

  There is also a large literature, primarily from the security community, on dynamic intrusion and anomaly detection, such as for web servers (e.g., [122, 123, 124, 125]). Intrusion and anomaly detection identify buggy behavior while the program is running. Non-webserver programs require other types of anomaly detection, such as methods that track other layers of the network stack, or that monitor system calls or library calls. Other approaches, such as instruction-set randomization [126] or specification mining [127] can also be expanded to report anomalies. In these systems, both false positives and false negatives are a concern, and researchers have developed techniques to intelligently retrain anomaly detectors in the face of software change [128].

- **A mechanism to predict or observe program behavior.** *Static* techniques predict behavior by analyzing program source code; *dynamic* techniques analyze actual behavior by compiling and running the source code on test inputs or indicative workloads. Static techniques tend to overapproximate possible behavior, leading more commonly to false positives; dynamic techniques tend to underapproximate possible behavior, leading to false negatives. Improvements along this axis seek to more accurately model the behavior of a system and then check it against more complex properties [129, 130].

- **A mechanism for compiling predicted deviations from correct behavior into an actionable report.** The output of a bug-finding tool is a report or set of reports of potentially buggy behavior, to be presented to developers for review and possible action [131]. Such a report may contain additional information to help a

human understand the reported defect. Examples include an execution trace or a candidate source code location associated with the bug [132]. This information may be further minimized [133] or explained [134]. This type of analysis has also been used to analyze crash reports to provide additional information to developers [32], separate from the problem of actually identifying bugs pre-crash.

An early and popular technique was embodied by the Lint program [135], which flagged suspicious behavior and code patterns in C code. Today, static bug-finding tools like FindBugs [30] and Coverity [33] have found commercial acceptance and success, the PreFix [136] tool is used internally by Microsoft, and the PreFAST tool ships worldwide with Microsoft Visual Studio [137].

Such tools strive to improve software quality and decrease software maintenance costs by identifying bugs in code as early as possible and simplifying the process of finding and fixing bugs. The output of a bug-finding tool is a report, that may be entered into a bug database (as described above) or otherwise presented to a developer for triage and repair. Ultimately, however, repairs for unannotated programs must still be made manually, by a human developer.

2.4 How do humans fix bugs?

A repair (or *patch* or *fix*) for a bug is a set of changes to the software source code such that the implementation no longer displays the undesired behavior or is brought more in line with its specification. To identify a suitable repair for a bug, a developer must understand first, the specification of the required behavior; second, the implementation that contains the defect; and third, where and how they differ. Debugging refers to the process of gaining these three pieces of knowledge about a particular bug and then identifying and validating a set of changes to the source code or configuration that fixes it. Ideally, such a fix only affects the undesired behavior while maintaining the other required behavior of the system, and avoids introducing new undesirable behavior (e.g., new security vulnerabilities or failure modes).

Programmers may use tools to step through and inspect the program execution leading to the bug (such as GDB [138]) or to gather more information about the nature of the failure (cf. Valgrind [139], which, among other purposes, is useful for debugging memory errors). They often must engage in complex reasoning about program structure and control-flow [140] to fully understand the problem.

Inspecting all of a program's source code or stepping through the entire execution of a typical software system is usually prohibitively expensive and time-consuming. This is especially true because most bug fixes are quite small, consisting of changes to 10 or 20 lines of code [141] even in systems that comprise millions of lines of code. Finding the appropriate location to change to affect a very particular piece of behavior can be like finding a needle in a haystack. Fault localization seeks to alleviate some of this burden by automatically identifying smaller sets of code likely to be implicated in a particular error, with the goal of reducing the amount of code that must be examined to identify a reasonable fix. The simplest of these techniques use intersection on sets of statements executed on buggy and

normal inputs to identify code statements whose execution is strongly associated with faulty behavior [142]. Other approaches use statistical measures of varying complexity to implicate lines of code [143] or logical predicates over program data or control flow [1] in a particular error; this active research area is broadly referred to as "spectrum-based localization" [10, 144].

Once the suspect code location has been identified, the developer may experiment with several different patches over the course of identifying one that works well. This involves changing the code and evaluating the effect of the change, typically by compiling and running the changed code on a set of indicative test cases. Even when a candidate set of changes does not correspond to the ultimate fix, the developer can learn about the undesirable behavior by evaluating the effect of source code changes on observed dynamic program behavior. The utility of such information is shown by the fact that defect reports that include a candidate repair for the defect in question are addressed and patched more quickly by developers than those without such patches, even when those patches are not what the programmer ultimately deploys (that is, even when the suggested patches are wrong) [145].

Ideally, a good repair for a bug should address the underlying behavioral issue, avoid introducing new faults or vulnerabilities, maintain existing functionality, remain consistent with the overall design goals of the system, and enable future maintainability. These goals are difficult to measure, let alone guarantee. For example, measuring software maintainability over time remains an active area of research [146, 147, 148]. Additionally, researchers have devoted attention to debugging assistance, aiming to produce information to help programmers understand and correct software errors [149]. Recent debugging advances include replay debugging [31], cooperative statistical bug isolation [1], and statically debugging fully-specified programs [150]. Other techniques mine program history and related artifacts to suggest bug repairs or otherwise provide debugging support [11, 151].

While developers do repair many bugs in practice, humans are error-prone, and a nontrivial proportion of fixes checked in by humans in practice have been observed to be "wrong", in that they do not fix the underlying issue, must ultimately be reverted or undone, or introduce new vulnerabilities or error modes into the source code [8]. Finding correct and high-quality fixes remains an open problem in both research and industrial practice. We review previous research techniques for automatic error preemption and repair in Section 2.6.

2.5 What is metaheuristic search?

Human debugging has many characteristics of a search problem, where the search objective is a set of changes that affect observed program behavior in a particular way. The search space ranges over possible patches that can be applied to a program. This space is infinite: a program can be trivially "patched" into any other program by applying a patch that deletes the source of the original and replaces it with a new one, and there are an infinite number of algorithms that can be encoded as programs [152]. Debugging must thus traverse this infinite space in an intelligent

way. The characterization of the patch-construction process as a search problem underlies the approach described in this dissertation. This section therefore provides background on metaheuristic search in general, and genetic algorithms and genetic programming in particular, as they form the core of our approach.

A metaheuristic search is an optimization method that iteratively generates and/or improves candidate solutions to optimize an external measure of quality [153]. Metaheuristic search strategies cannot guarantee an optimal solution for infinite search spaces; rather, they seek to find "good" solutions. Such strategies are thus typically used to operate over large, complex, multi-dimensional spaces, especially those that are poorly-handled by traditional optimization techniques. The term is often used to refer to a large subclass of the strategies that comprise stochastic optimization methods, which in turn are optimization methods that incorporate randomness.

Designing or using such a strategy requires definitions for how candidate solutions are represented (what does a candidate solution look like computationally?), how they are generated (using randomness or not) either from scratch or via changes to an existing candidate, and how to measure a candidate's suitability relative to the objective of the search. These choices define the type of solution that can be found as well as how the search progresses, and in particular how it balances *exploration* of the space and *exploitation* of previously-evaluated good solutions. The difference between the two concerns is well-illustrated by the contrast between *random search* and *hill climbing*, which lie at the two extremes. A random search focuses entirely on *exploration*: it generates and evaluates random candidate solutions, returning the best found once a time limit has been reached [154]. A hill climbing search starts with a random candidate solution. It then makes a random change to that candidate to generate a new candidate. If the new candidate is better than the original, the new candidate is kept, and the search continues by modifying it. If not, the original is kept, and a different random change is attempted. Hill climbing thus focuses significantly on *exploitation*, or making use of the best-known solution found to date.

While both types of search are well-suited to particular types of problems, focusing on the extremes of either exploration or exploitation can limit success. Random search fails to take into account any previously-acquired knowledge about what a good solution might look like and can reduce to sampling the search space, which is often unacceptably inefficient. Hill climbing can get stuck in local optima, because it relies entirely on the currently-best-known solution to define where it should next direct the search.

More advanced search strategies thus are often designed to trade off these two concerns, using exploitation to guide the search in the direction of a good solution by leveraging knowledge about previous good solutions, and exploration to help avoid local optima. Metaheuristic search is an active area of research [155], and there are a number of metaheuristic search strategies designed for different application areas (e.g., scatter search [156] or particle swarm optimization [157] for continuous problem domains). Well-known examples of such strategies include random search, hill climbing, simulated annealing [158, 159], ant-colony optimization [160], and genetic algorithms [36].

One class of search methods that explicitly balances exploration and exploitation is known as genetic algorithms

(GA). A genetic algorithm is a stochastic optimization strategy that mimics parts of the process of biological evolution; such algorithms can search complex, high-dimensional spaces.

A GA is an iterative population search, in that it iterates over a set of candidate solutions, moving them collectively towards a goal by both randomly mutating single existing solutions and combining multiple existing solutions to generate new candidates. Each individual solution is comprised of a set of properties (its genome, or chromosome). In practice, candidate solutions in GAs are often represented as bit- or character-arrays, with each element in the array corresponding to a feature in the solution space. A characteristic of the type of solution space that a GA can search is that each feature in a candidate solution should be independently modifiable, and that the different sets of properties can be reshuffled or recombined with other sets of properties. These modifications are used to generate new candidate solutions from existing candidates, and are performed by two fundamental operators: the mutation operator, which randomly changes one of the features in the property set of an individual genome; and the crossover operator, which recombines the genomes of two independent solutions to produce new candidate [42]. The mutation operator simulates the random mutations that can affect individual genes in biological DNA; crossover simulates the recombination of DNA when individuals reproduce in biology.

A GA then iterates over the population to generate new populations, which are also referred to as generations. It therefore begins by generating a random population of candidate solutions as a starting point. As an optimization problem, a GA includes an objective function that is being maximized. Each individual in the current population is evaluated to measure its desirability according to this function, which is often called a fitness function in this application. The fitness of a candidate often influences the probability that it "reproduces" (is selected as a "parent" to be used in crossover) or that it is retained in the next generation.

The two basic operators (mutation and crossover) enable both *exploration* (mutation explicitly introduces randomness; crossover can enable exploration between parents far separated in the search space, and is sometimes thus viewed as a macro-mutation operator) and *exploitation* of previously-known solutions (again, via mutation, which introduces randomness in existing candidate solutions, as well as crossover, which combines and retains information from previously-identified candidates). The probabilities with which individuals are mutated, crossed over, selected, and propagated to the next generation influence how much exploration is favored as compared to exploration, and these parameters are typically tunable.

Genetic programming is the application of genetic algorithms to source code, typically seeking to discover computer programs tailored to a particular task [41, 42]. In GP, the chromosomes are typically tree-based representation of a program. The independent features in the genomes of a GP search are typically groups of statements corresponding to source code. The mutation and crossover operators are often tree-based, as they apply to tree-represented source code.

GenProg uses genetic programming to search through the space of candidate patches to a given program (rather than to search through the space of candidate programs); more detail is given in subsequent chapters. It is worth noting,

however, that although GP has solved an impressive range of problems (see, for example, [43]), most program repair GP efforts to date have been plagued by scalability concerns [15]. There has been considerable recent interest in and independent validations of the potential of GP or random mutation for repair [161]. It has been applied to new languages [162, 163, 164] and domains [165, 166], and has improved non-functional program properties, particularly execution time [167].

In general, however, these previous attempts [44, 168] are limited to very small, hand-coded examples. Some previous efforts to apply GP to program repair [169] relied on formal specifications to evaluate fitness, which is time-intensive and limited in applicability, though the idea of co-evolving tests and code has since been expanded [170]. Our work is the first to report viable results on the repair of real defects in real programs, which is the dominant concern in this dissertation.

2.6 What are automatic ways to fix bugs?

Although manual effort remains the state-of-the-art for searching for bug fixes in industrial practice, the number and cost of existing bugs have motivated research towards automatically mitigating the effects of defects and vulnerabilities in software systems. The goal of such approaches is to either catch faulty behavior just before or as it happens and mitigate its harmful effects, transform the program or execute it in a way that precludes certain types of faulty behavior, or generate patches at the source- or binary-level that either prevent entirely a class of bugs or fix a particular bug.

In this context, the definition of correct or fixed behavior varies. Techniques that rely on verification define correctness through soundness with respect to a specification [171]. Others concern themselves with one specific type of error, typically buffer overruns, and define correctness with respect to the possibility of such an error in practice. Some techniques disregard the notion of formal, provable correctness entirely, striving to achieve what has been called *failure-oblivious computing* [24]: that is, continued execution in the face of known errors. Returning zero, for example, for an invalid memory access is often preferable to wholesale program failure, at least from the perspective of the user.

One class of approaches to automatic error preemption and repair seeks to dynamically detect and prevent harmful effects from particular types of errors. Programs with monitoring instrumentation can detect data structure inconsistency or memory over- or under-flows. Demsky *et al.* [25] automatically insert run-time monitoring code to detect if a data structure ever violates a given formal consistency specification and modify it back to a consistent state, allowing buggy programs to continue to execute; various other strategies have been deployed to similar effect [172]. Smirnov *et al.* [173, 174] automatically compile C programs with integrated code for detecting overflow attacks, creating trace logs containing information about the exploit, and generating a corresponding attack signature and software patch. DYBOC [12] instruments vulnerable memory allocations such that over- or underflows trigger exceptions that are addressed by specific handlers. Along similar lines, selected transactional emulation [175], executes potentially

vulnerable functions in an emulation environment, preventing them from damaging a system using pre-specified repair approaches; a more accurate approach uses rescue points [176]. Jolt [35] assists in the dynamic detection of and recovery from infinite loops.

Other research efforts (including our own) focus more directly on patch generation. Keromytis *et al.* [177] proposed a system to counter worms by using an intrusion detector to identify vulnerable code or memory and preemptively enumerating repair templates to generate candidate patches. At the binary level, Clearview [13] uses runtime monitors to flag erroneous executions and then identifies invariant violations characterizing the failure, generates candidate patches that change program state or control flow accordingly, and deploys and observes those candidates on several program variants to select the best patch for continued deployment. AutoFix-E [171] uses contracts present in Eiffel code to propose semantically sound fixes. Gopinath *et al.* also use formal specifications to transform buggy programs [178]; PACHIKA [179] infers object behavior models to propose candidate fixes. Axis [180] and AFix [181] soundly patch single-variable atomicity violations;

These and similar techniques have several drawbacks. First, they are almost exclusively designed to address particular types of defects, making use of pre-enumerated repair strategies or templates, or requiring formal program specifications. In practice, although specifications enable semantic soundness guarantees, despite recent advances in specification mining (e.g., [121, 182]), formal specifications are rarely available. Moreover, most specification languages or formalisms are limited in the types of errors they can define. Although some of the non-specification based techniques are theoretically applicable to more than one type of security vulnerability, typically, evaluations are limited to buffer over- and underflows. The exception to this rule, Clearview, has been shown to also address illegal control-flow transfers [63], but is limited by the availability of external monitors for any given vulnerability type. Generality thus remains a concern that has strongly motivated the work presented in this dissertation. Additionally, concurrency bugs remain a significant challenge. AFix and Axis are two of the only techniques to date to address them explicitly, although several of the other techniques (including the work presented in this dissertation) can repair deterministic bugs in multi-threaded programs.

Second, these techniques require either source code instrumentation (e.g. Smirnov [173, 174]), Demsky [25]), which increases source code size (by 14% on Apache in DYBOC [12]), run-time monitoring (e.g.,DYBOC, Clearview [13], Keromytis *et al.* [177]), or virtual execution (Clearview, selected transactional emulation [175]), imposing substantial run-time overhead (20% for DYBOC, up to 150% for Smirnov, 73% on Apache for selected transactional emulation, 47% on Firefox for Clearview and, in general, at least the run-time cost of the chosen monitors). Scalability thus remains a significant concern in automatic repair research.

2.7 Summary

This chapter presents background and an overview of related work on bugs in software and how they can be avoided, identified, and repaired by humans; metaheuristic search strategies, specifically genetic algorithms and genetic programming; and automatic error preemption and repair. We established in Chapter 1 that bugs in software represent a critical economic and time cost in the software development process. This dissertation proposes to combine metaheuristic search strategies with insights from software engineering and testing to automatically repair software bugs. We present our approach in detail in the next chapter.

Chapter 3

GenProg: automatic program repair using genetic programming

THIS chapter describes *GenProg*, a method that automatically fixes bugs in off-the-shelf programs. The algorithm is designed to exhibit the three important properties of a viable, real-world, and human competitive automatic repair solution, as articulated in Chapter 1:

- **Expressive power.** We consider an automatic program repair solution expressive when it can repair a variety of different types of bugs in a variety of different types of programs, and when it can repair a non-trivial proportion of high-priority bugs that humans developers encounter in practice.

- **Repair quality.** A repair is of high quality if it fixes the underlying cause of a defect (instead of simply masking the symptoms) while maintaining the program's other required functionality and avoiding the introduction of new vulnerabilities or defects.

- **Scalability.** We say that an automatic program repair solution is scalable when it can repair bugs in large, indicative real-world legacy systems comprising thousands to millions of lines of code at a feasible (or human competitive) time- and monetary cost.

Given a program with a bug that is exposed by at least one test case, GenProg produces a patch for that bug by searching for a set of changes that addresses the incorrect behavior while maintaining other required program behavior and avoiding the introduction of new defects or vulnerabilities. GenProg finds such patches using stochastic search: it selectively enumerates random candidate patches and evaluates them to determine if they are valid.

This description raises several important algorithmic design questions:

- **What types of patches does GenProg generate?** GenProg composes and combines small changes to the input program to produce candidate patches. The types of changes that GenProg considers are broadly designed so as to allow maximal expressivity. GenProg intelligently constrains the search space by targeting the changes to portions of the source code that are strongly correlated with the defect, to increase the probability that a candidate random patch influences the behavior of the program appropriately.

- **How are candidate patches enumerated?** GenProg uses genetic programming (GP) to generate, recombine, and iterate over candidate changes. GP is well-suited for this domain, as we argue in Section 3.3, but the nature of the program repair problem requires novel design decisions for the solution representation and mutation operators that define the search.

- **How are candidate patches evaluated?** A GP search requires a fitness function to evaluate the desirability of candidate solutions. Because the search problem at hand is for a patch that affects a program's dynamic behavior, GenProg uses test cases as proxies for formal correctness specifications that encode program requirements.

The rest of this chapter gives details about and justification for our algorithmic design decisions. We begin by presenting an illustrating example, adapted from a real vulnerability, that we develop throughout the chapter to illustrate the concepts herein (Section 3.1).

We define a valid repair in the context of the automatic search (Section 3.2); this serves to define the input/output contract of the algorithm. We provide an explanation of and justification for the use of genetic programming in this application (Section 3.3). The creation of a new GP algorithm involves several design decisions, which we elucidate thereafter (Section 3.4). First, we design a suitable *representation* for candidate solutions (Section 3.4.1). We specify how to generate the initial population and how to select individuals for continued iteration (Section 3.4.2). We then define operators for crossover and mutation (Section 3.4.3) to recombine and generate new variants from previously-identified candidate solutions.

The mutation operators motivate a discussion of the program repair search space, which we constrain and manage in novel ways (Section 3.5). In particular, we target the random changes that comprise candidate solution patches to areas of the code that are most likely to be suitable locations for repair, taking advantage of insights from previous fault localization research.

The final important GP component we define is the fitness function (Section 3.6), which evaluates the suitability of intermediate variants (guiding their selection) and defines the stopping criterion for the search.

GP algorithms are prone to code bloat [15], that is, the generation of unnecessary extraneous portions of individual genomes (or introns) that do not contribute to the solution but that complicate and unnecessarily candidate solutions. We managed code growth post-facto, in a *minimization step*, which we also explain in this chapter (Section 3.7).

The main contribution of this chapter is GenProg, a novel algorithm for repairing bugs in real-world programs. More specifically:

- A genetic programming technique that efficiently searches the space of patches for a buggy program.
- A compact, scalable, and expressive representation of candidate solutions.
- Efficient and domain-specific crossover, mutation, and selection operators for manipulating and traversing the space of candidate solutions.
- A formalization of the program repair search space, including the novel application of fault localization to this domain and the introduction of the idea of fix localization. This conception of the search space is novel on its own, and also enables the algorithm's scalability by intelligently constraining the search.
- A technique to efficiently yet correctly guide the GP search for a repair by sampling test suites.
- A novel application of software engineering debugging and tree-based differencing techniques to overcome genetic programming's classic code bloat problem as it manifests in this application.

In subsequent chapters, we empirically substantiate our claims about GenProg's expressive power, output quality, scalability, and human-competitiveness; analyze GenProg's scaling and performance behaviors; and directly and comparatively evaluate representation and operator choices as well as the automatic repair search space.

3.1 Illustrative example

For the purposes of clarifying this chapter's presentation and motivating important insights underlying our approach, we present as a running example a defect adapted from a publicly-available vulnerability report. The defect in question appears in version 0.5.0 of `nullhttpd` webserver [183], a lightweight multithreaded webserver that handles static content as well as CGI scripts. This bug is useful as an illustrative example because it represents a real security defect, which are particularly important to repair and which are prevalent in practice, and thus strongly motivate the work in this dissertation on the whole. Additionally, a number of the insights that inform our algorithmic design decisions are based on our intuitions about the nature of real software and defects in practice, and these characteristics are particularly clear in this bug. As a result, it serves as an elaborated worked example to clarify the subsequent formal algorithmic presentation.

We present a snippet of code adapted from the larger program source (which totals 5575 lines of C code). We describe the intended functionality of the code, the mistake in the implementation that leads to a security vulnerability, and a candidate solution for that vulnerability that corresponds to what the human developer did to repair it.

```
1  char* ProcessRequest() {
2    int length, rc;
3    char[10] request_method;
4    char* buff;
5    while(line = sgets(line, socket)) {
6      if(line == "Request:")
7        strcpy(request_method, line+12)
8      if(line == "Content-Length:")
9        length=atoi(line+16);
10   }
11   if (request_method == "GET")
12     buff=DoGETRequest(socket,length);
13   else if(request_method == "POST") {
14     buff=calloc(length, sizeof(char));
15     rc=recv(socket,buff,length)
16     buff[length]='\0';
17   }
18   return buff;
19 }
```

(a) Buggy webserver code snippet.

```
4    ...
5    while(line = sgets(line, socket)) {
6      if(line == "Request:")
7        strcpy(request_method, line+12)
8      if(line == "Content-Length:")
9        length=atoi(line+16);
10   }
11   if (request_method == "GET")
12     buff=DoGETRequest(socket,length);
13   else if(request_method == "POST") {
14 +   if (length <= 0)
15 +       return null;
16     buff=calloc(length, sizeof(char));
17     rc=recv(socket,buff,length)
18     buff[length]='\0';
19   }
20   return buff;
21 }
```

(b) Patched webserver.

Figure 3.1: Pseudocode of a buggy webserver implementation, and a repaired version of the same program.

Consider the pseudocode for a part of a webserver, shown in Figure 3.1(a). The purpose of a webserver is to deliver content using the Hypertext Transfer Protocol. Its primary function is thus to receive requests from external users on a network socket, process the request, and respond to it appropriately. In the pseudocode, function `ProcessRequest` processes an incoming request to the webserver based on data copied from the request header. Note that on line 14, the call to `calloc` to allocate memory to hold request content uses the content length provided by a POST request, as copied from the header on line 8. The content length is provided by the remote user who specified the request, and by using it unchecked, this version of `nullhttpd` is implicitly trusting that user to provide a value that is valid for the purposes of allocating memory. A malicious attacker can provide a negative value for `Content-Length` and a malicious payload in the request body to overflow the heap and kill or remotely gain control of the running server [183].

This is an example of a buffer overflow vulnerability caused by improper input validation as well as improper memory allocation: the webserver does not check that the user-provided input is suitable for the purposes of allocating memory before using it to do so. It can be repaired fairly simply by adding a check on the content length before using it in the call to `calloc`, which is exactly what the human developer did to patch the bug. This candidate patch, applied to the original program, is shown in Figure 3.1(b).

In subsequent sections, we periodically refer back to this example to illustrate the algorithmic presentation and its underlying design insights. GenProg can also successfully repair the defect on which this example is based, as we describe in Chapter 4. At a high level, the goal of GenProg is to get from Figure 3.1(a) to Figure 3.1(b) automatically.

3.2 What constitutes a valid repair?

In this section, we define the parameters of GenProg's input/output contract. We describe what GenProg takes as input, how it expects a bug to be defined and described, and what it looks for as a valid repair. This in turn defines the goal of the GenProg search for a patch to a buggy program.

GenProg takes as input program source code (written in C) that contains a defect.[1] GenProg also requires a specification of the defect to repair. As described in Chapter 2, program defects are found and reported in myriad ways, and often a defect report will contain steps to reproduce, an input that corresponds to an unexpected output, or an executed or predicted trace of program behavior corresponding to the buggy behavior. Humans take a variety of approaches to create and validate candidate fixes for a given bug, and the different options present tradeoffs in terms of the guarantees they provide versus the scalability and applicability of the technique. The two basic ways that humans establish the correctness of a patch in non-safety-critical applications are code review and individual human reasoning and ingenuity, or observed dynamic behavior on a test case or workload that exercises the code and expresses an expectation of correct behavior.

Because we seek an automatic technique that alleviates some of the human burden of code repair and scales in terms of time complexity, we choose to avoid relying on human interaction to determine if a program's behavior is or remains buggy. In theory, a technique could enumerate patches and present them individually to an expert developer to ask if they are acceptable. However, as established, the human debugging process is time-consuming, and involves complex reasoning, and thus this approach would not scale.

As a result, GenProg relies on the second approach that industrial practitioners take to establishing correctness: test cases. We codify desired behavior by running the program on an input, defining the expected output on that input, and defining a mechanism that determines whether the observed behavior matches the expected behavior. To illustrate by referring back to our running example in Figure 3.1(a), we can write a test case that demonstrates the bug by sending a POST request with a negative content-length and a malicious payload to the webserver in order to crash it, and then checks the webserver to determine if it is still running. Unmodified **`nullhttpd`** fails this test case.

However, GenProg needs more input to fully define expected program behavior. Defining desired program behavior exclusively by what we want **`nullhttpd`** to *not* do (crash on a POST request with a negative content length) may lead to undesirable results. Many changes that are "acceptable" according to this one test would not be acceptable from a whole-system perspective. Consider the following variant of **`nullhttpd`**, created by a patch that replaces the body of the function with a **`return null`**:

```
1  char* ProcessRequest() { return null; }
```

[1] The implementation we describe in this dissertation accepts C source code, and thus the results we present in subsequent chapters focus on programs written in C. We note, however, that GenProg and extensions to GenProg produced by other researchers have been successfully applied to other languages, such as Java [184] and assembly code [163]; our focus on C is an implementation detail rather than an intrinsic restriction.

This version of `ProcessRequest` does not crash on the bug-encoding test case, but also fails to process any requests at all. The repaired program should pass the error-encoding test case, but it must also retain core functionality to be considered acceptable. Such functionality can also be expressed with test cases, such as a regression test case that obtains `index.html` and compares the retrieved copy against the expected output.[2]

Thus, in addition to the program source code, GenProg also takes as input a set of test cases, whether automatically generated or hand-written. At least one test case should be failing; at least one, and typically several, should be passing. We call the failing test case the negative test case, and it encodes the defect under repair. We call the passing test cases the positive test cases that describe other functional and non-functional program requirements. Both types of test cases—initially passing and initially failing—are typically required to guide the search to an adequate repair.

The output of the tool is a set of one or more code changes that, when applied to the program, cause it to pass all of the test cases. We refer to these changes as a patch or repair interchangeably. This definition has been followed by other researchers in the bug preemption and repair literature, notably Rinard *et al.* [13]. We evaluate its implications for repair quality in Chapter 4. We note that while removing humans from the repair loop presents challenges, humans themselves are imperfect when repairing defects [8]. Measuring and guaranteeing repair quality remains an open research challenge, motivated as much by the challenges of human debugging activities as by research in automatic repair.

3.3 Why Genetic Programming?

In this section, we describe the mechanism GenProg uses to enumerate and search through candidate patches to a buggy program. The goal of our algorithmic design is to develop an effective search through candidate repair patches that is scalable, expressive, and correct.

There are three reasons that Genetic Programming (GP) is uniquely well-adapted to this search problem. First, evolutionary algorithms such as GP perform well in complex and rugged search landscapes, even those that include many local optima [185]. The combination of crossover and small local mutations provides a robust search mechanism that effectively trades off exploitation and exploration. For example, crossover allows individual intermediate candidate solutions to combine partial solutions, transferring knowledge between successful candidates and avoiding local optima [36]. Second, a specific implication of this feature is that GPs (and GAs in general) are especially tolerant of a noisy fitness signal [186]. This is important because approximating partial program—and thus partial patch—correctness is difficult, especially in the absence of full verification. Instead, we use test cases as a fitness signal (see Section 3.6), which are scalable and readily available. However, test cases provide a coarse indication of partial correctness, because "passes a test case" is a binary notion that does not consider intermediate program behavior, only its result. Finally, GP

[2] In practice, we use several test cases to express program requirements; we describe only one here for brevity.

is intended to operate on programs as tree-structured data [42], a representation choice which affords natural advantages in this application (see Section 3.4.1).

The GP we use here differs from most previous work in genetic and evolutionary algorithms, as we have adapted it in several key ways to the problem of automatically repairing bugs in software systems of a realistic size. Notably, while many previous applications of GP evolve a solution from nothing or from an entirely random generation of initial individuals, the initial solution in our application is much closer to the eventual answer: even buggy programs are mostly correct, and patches are typically small [187, 188]. As a result, the underlying representation and operators used in our GP implementation, and many of the experimental parameters we use when running GenProg, vary considerably from those used in previous applications (e.g., other applications use much larger population sizes and run their algorithms for many more iterations than we do [189]). We provide explanation and justification for those choices in subsequent sections, and evaluate them in in Chapter 6.

Figure 3.2 shows an architecture diagram of GenProg to illustrate the primary details of the iterative search algorithm, and to identify the components of the algorithm that are discussed in subsequent sections. It therefore serves as a partial road map for the rest of this chapter (not shown are the pre- and post-processing steps, also discussed later in this chapter). GenProg takes as input C source code containing a defect and a set of positive and negative test cases (Figure 3.2(a), see Section 3.2). The goal of the search is to identify a candidate patch that, when applied to the original program, produces a new program that passes all the input test cases.

GenProg starts by generating an initial population of patches (Section 3.4.1) to apply to the input program based on the input program, test cases, and preprocessing. This forms the first of the generations over which the algorithm operates (Figure 3.2(b), Section 3.4.2). Each candidate patch is evaluated for its fitness by applying it to the initial program and compiling and running the result on the test cases (Figure 3.2(c)). Individuals are selected with a probability based on their fitness to be propagated into the next generation. Lower-fitness individuals have a lower probability of selection and thus may be discarded (Figure 3.2(d)). Retained individuals are mutated and recombined using mutation and crossover operators (Figure 3.2(b), Section 3.4.3).

The process iterates until resources are exhausted or a repair is found; its output in this case is a patch (Figure 3.2(f)). A deterministic post-processing step can minimize the patch beyond what is first found (Section 3.7). The first patch identified is the initial repair; the minimized patch is the final repair. Because the search is stochastic and the space of candidate patches is infinite, GenProg is typically run with a preset timeout of a certain number of iterations or execution time, and several different random runs of the algorithm are typically executed in parallel.

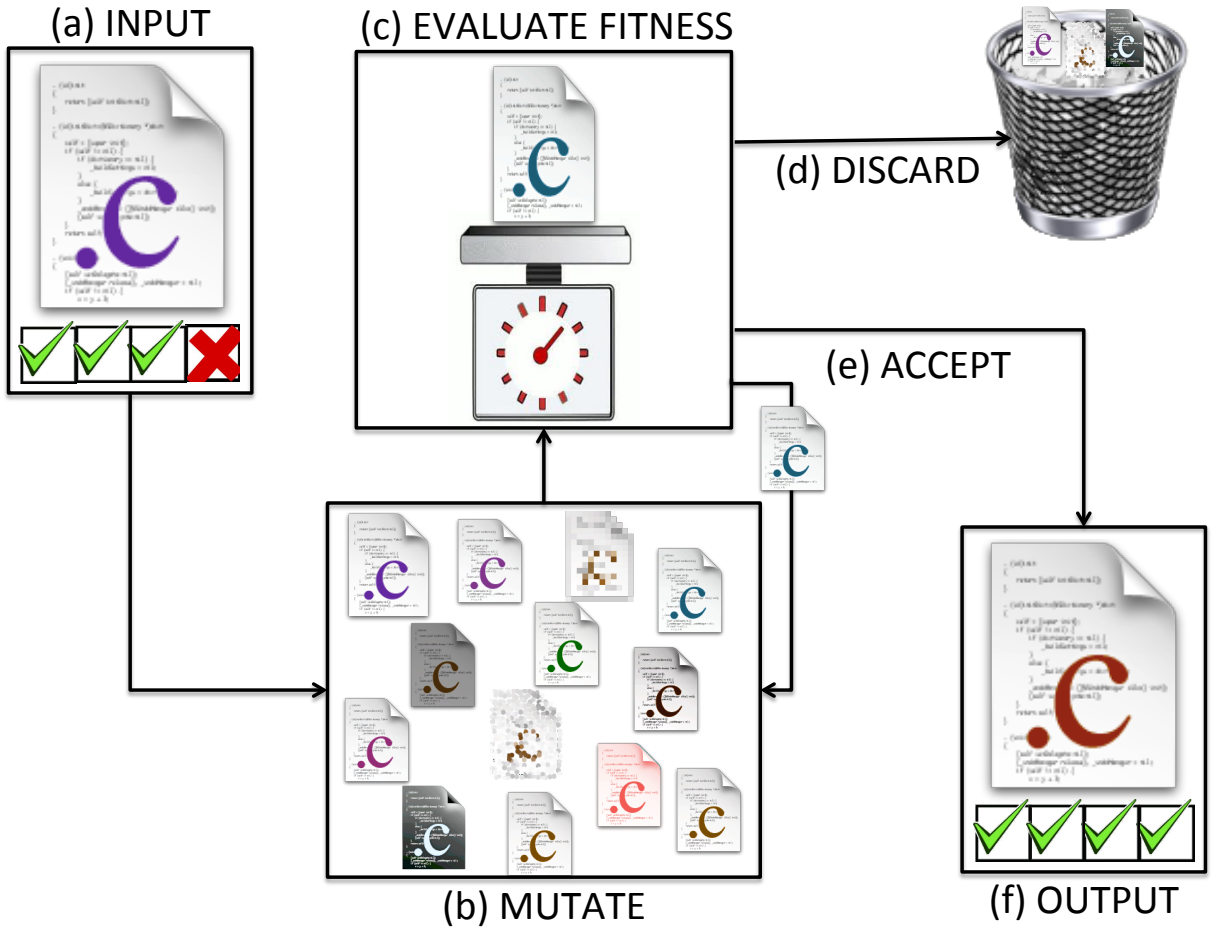

Figure 3.2: High-level architecture of the GenProg algorithm. GenProg accepts C source code and a set of test cases, at least one of which is failing, as input (a). It iterates over a population of candidate solutions, produced by random mutation and recombination of existing candidate solutions (b). Candidate solutions are evaluated for fitness, or desirability (c). The fitness of a candidate solution influences the probability that it is discarded (d) or selected and retained for continued iteration. The goal of the search is a solution (a patch) that causes the input program to pass all of its test cases (f).

Input: Full fitness predicate FullFitness : $Patch \to \mathbb{B}$
Input: Sampled fitness SampleFit : $Patch \to \mathbb{R}$
Input: Mutation operator mutate : $Patch \to Patch$
Input: Crossover operator crossover : $Patch^2 \to Patch^2$
Input: Parameter $PopSize$
Output: Patch that passes FullFitness

1: **let** $Pop \leftarrow$ map mutate over $PopSize$ copies of $\langle\,\rangle$
2: **repeat**
3: **let** $parents \leftarrow$ tournSelect$(Pop, Popsize,$ SampleFit$)$
4: **let** $offspr \leftarrow$ map crossover over $parents$, pairwise
5: $Pop \leftarrow$ map mutate over $parents \cup offspr$
6: **until** $\exists\, candidate \in Pop$. FullFitness$(candidate)$ **return** $candidate$

Figure 3.3: High-level pseudocode for the main loop of our technique. crossover, mutate, and FullFitness and SampleFit are discussed in subsequent sections.

3.4 Core genetic programming algorithm

We formalize the architecture overview in Figure 3.2 as algorithmic pseudocode in Figure 3.3. The first important choice in designing a new GP concerns how the algorithm represents candidate solutions, which correspond to the individuals in the population. We represent a candidate $Patch$ as a sequence of abstract syntax tree edit operations to the input program, parametrized by tree node numbers. We explain and justify this choice in Section 3.4.1.

The search begins by constructing and evaluating a population of random patches, and line 1 of Figure 3.3 initializes the population by independently mutating copies of the empty patch (Section 3.4.2). Lines 2–6 correspond to one iteration or generation of the algorithm. A GP iterates by selecting high-fitness individuals to copy into the next generation (line 3, Section 3.4.2) and introducing variations with domain-specific mutation (`mutate`) and crossover (`crossover`) operators (lines 4 and 5; Section 3.4.3). "Crossover" corresponds to genetic crossover events, where parents are taken pairwise at random to exchange pieces of their representation; two parents produce two offspring. Each parent and each offspring is mutated once (via genetic "mutation") and the resulting individuals form the incoming population for the next iteration. Operator design therefore comprises the second key choice in our new GP algorithm, and we discuss it in Section 3.4.3. An important innovation in our approach is a set of intelligent constraints on the search space, which in turn constrain the application of those operators; we discuss the program repair search space in Section 3.5.

Finally, we must also define how to evaluate the desirability of a candidate, because a candidate's fitness affects the probability that it is selected to contribute to the subsequent generation. In the pseudocode, the functions `SampleFit` and `FullFitness` evaluate variant fitness or desirability (lines 3 and 6) by applying candidate patches to the original program to produce a modified program that is evaluated on test cases (Section 3.6).

This cycle repeats until a patch is found or a pre-determined resource limit is consumed. We refer to one execution of the algorithm described in Figure 3.3 as a *trial*. Multiple trials are often run in parallel, each initialized with a distinct random seed.

3.4.1 Program Representation

This section describes and justifies the GenProg's representation of candidate solutions. At a high level, a solution in our application is a patch to the source code of the input program. The question is thus how to represent a candidate set of changes internally such that GenProg can meet its design goals. First, the representation should enable scalability to real-world software systems. It is ideally compact and should grow sub-linearly with the size of the program under repair. Second, it should be suited to describing valid changes to those programs. Put more formally, it is advantageous in evolutionary search [190] that there exists a *bijection* between the genotype (the individual candidate solutions expressible by the representation) and the phenotype (in this case, patched programs) of the candidate solutions. It is therefore beneficial if our representation is unlikely to describe *invalid* source code changes, such as changes that will

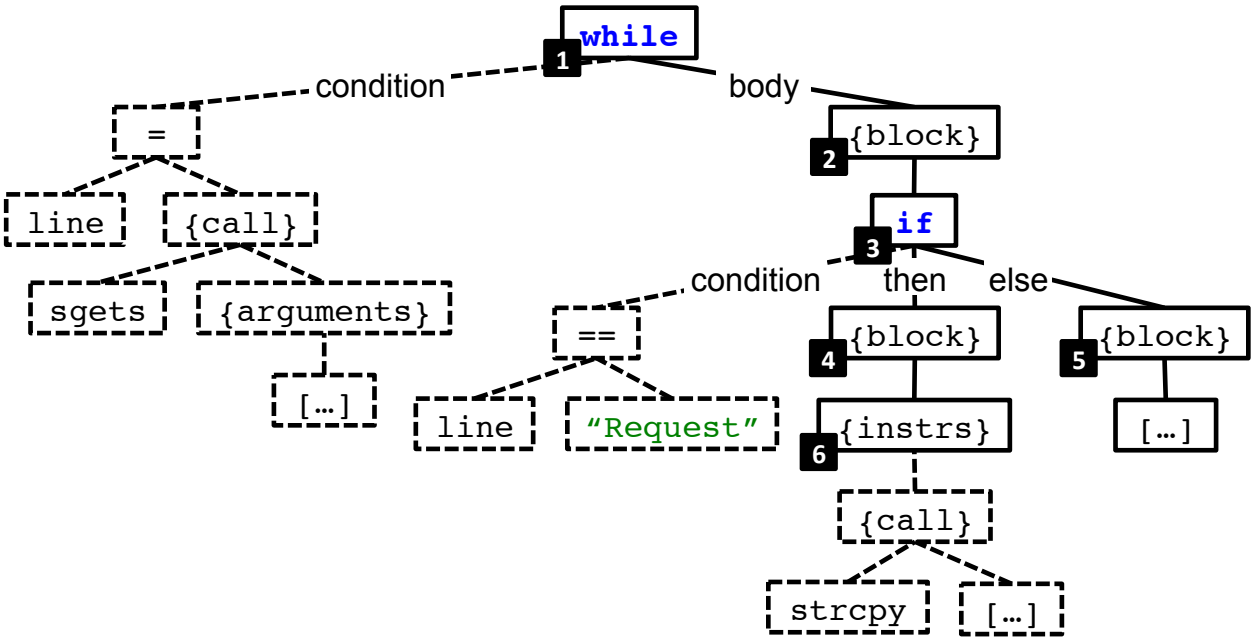

Figure 3.4: A tree-based representation of the abstract syntax tree of lines 5–9 of the example buggy code in Figure 3.1(a). The solid boxes correspond to the C *statements* that GenProg-generated patches may manipulate. They have been numbered via breadth-first traversal of the tree. The numbering scheme is arbitrary; the important point is that the nodes are identified uniquely. Dashed boxes in this diagram correspond to C *expressions*, which GenProg patches will not manipulate (nor declarations nor definitions). The “instrs” box (statement 6) corresponds to a list of *instructions* without control-flow; a GenProg patch may manipulate the list of instructions as a unit, but will not manipulate the individual components of the instructions themselves.

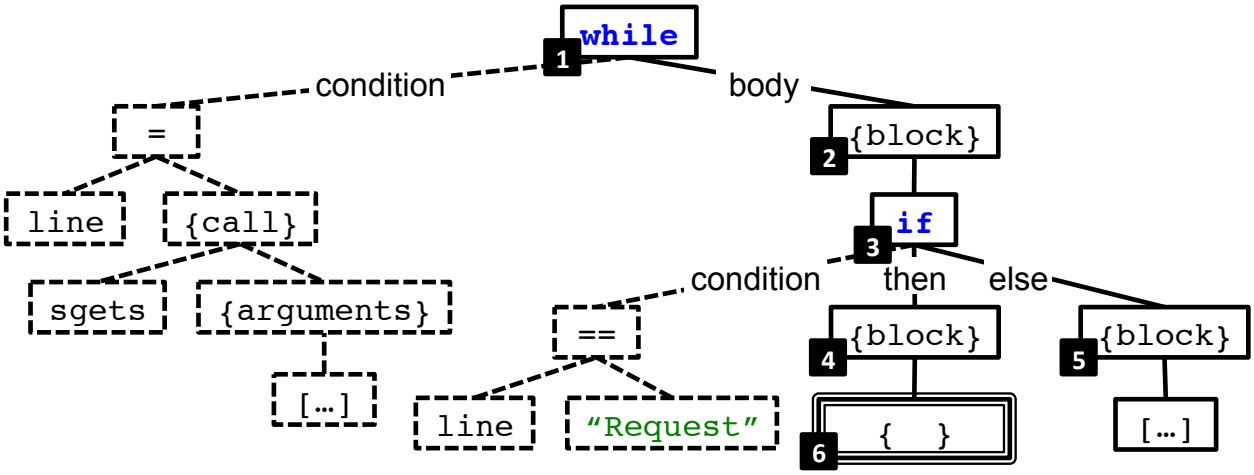

Figure 3.5: The result of applying the tree-based edit operation “Delete(6)” to the abstract syntax tree in Figure 3.4. Note that the entire subtree at statement 6 has been replaced with an empty block (shown in the decorated box); the original function call will not appear in the printed source code for this AST. The modified statement 6 may be manipulated by later edits, because the numbering scheme is unmodified.

result in a program that does not compile. A change can break compilation [191] at the lexer stage, e.g., by introducing a spelling mistake to a program key word; at the parsing stage, e.g., by introducing an unbalanced parenthesis; or at the semantic analysis stage, e.g., by moving a variable out of scope. We want a representation that precludes as many of

these types of errors as possible. Finally, the representation should be as expressive as possible within the bounds of scalability concerns, in that it should be able to describe many different types of changes to program code.

To meet these goals, GenProg explores the space of *tree-based edits* to the input program, also represented as a tree. Tree operations are well-studied in GP, and by operating on the program as a source tree, GenProg can explore changes that are syntactically (in terms of lexing and parsing [191]) and largely semantically valid. Tree-based edits also more naturally express the types of changes that humans perform on source code: human edits tend to manipulate program subtrees, an insight that underlies work in language-sensitive editors [192, 193].

The natural first step of the algorithm is thus to parse the input program into a tree. There are a number of commonly accepted tree structures for representing programs, such as control flow graphs (CFGs) and abstract syntax trees (ASTs) [191]. GenProg manipulates a program's abstract syntax tree (AST). An abstract syntax tree is a graph-based representation of the syntactic structure of source code. Each node of the tree corresponds to a programming language construct in the code [14], and relations between nodes express syntactic relationships between the constructs in the code. The tree is *abstract* in that it does not represent every detail that appears in the initial code, e.g., some bits of concrete syntax are omitted based on the language grammar. ASTs are efficient[3] and sufficiently powerful to represent all structured programs. Conversely, most abstract syntax trees can be reified as valid source code. GenProg generates a program's AST using the off-the-shelf CIL toolkit [194].

To illustrate, Figure 3.4 shows part of an abstract syntax tree of the code in Figure 3.1(a) (lines 5–9 of the code snippet). Note that components like parentheses, brackets, and semicolons that would be present in a parse tree (or concrete syntax tree) are omitted from the abstract syntax tree, but can be trivially reinserted to convert the tree back to source code. A tree-based edit manipulates subtrees, so "Delete(6)" in Figure 3.4 would delete the node and its subtrees, resulting in the tree shown in Figure 3.5. We describe the types of edits that GenProg performs in Section 3.4.3.

ASTs express program structure at multiple levels of abstraction or granularity, which raises a question of which types of nodes in the tree GenProg patches should manipulate. Although an AST has significantly fewer nodes than a source code file has characters, in a non-trivial program, there are still many candidate nodes in an AST that could be manipulated by a tree-based edit. We would like candidate patches to be expressive, in that we would like them to be able to express many different types of changes in behavior, while keeping the number of locations in the code that can be changed manageable.

GenProg-generated patches therefore operate on the constructs that the C language specification defines as *statements*, which includes all instructions (i.e., assignments and function calls), conditionals, blocks, and looping constructs [195]. GenProg patches therefore do not directly modify *expressions*, such as "`(1-2)`" or "`(!p)`", nor do they ever directly modify low-level control-flow directives such as `break`, `continue` or `goto`. This genotype representation reflects a

[3] In the worst case, any context-free grammar can be parsed into an AST in $\mathcal{O}(n^3)$ time. In practice, languages such as C can be parsed in near-linear time using optimized techniques such as LALR(1). [191]

tradeoff between expressive power and scalability: statement-level edits are sufficiently powerful in practice to address a variety of real-world bugs, as we substantiate in later chapters. By contrast, adding expressions as mutable units increases the search space between 2 and 10 times over mutating statements alone.

These decisions address how candidate patches should manipulate and represent the input *program*. However, the question about how to represent, create, and modify the candidate patches *themselves* remains. We could frame the search problem in the more traditional GP way by representing each candidate as a full program AST; the GP would then search for a version of the program that does not contain the bug in question. However, this representation choice grows linearly with the size of the input program and does not scale to real-world systems, especially in a modern commodity cloud compute setting, the likes of which industrial practitioners are increasingly using for testing and software development [89]. We see this scalability problem in our own experiments: for at least one-third of the scenarios in our largest benchmark set (Table 5.1), designed to be indicative of real-world systems, an initial population of variants represented as full ASTs do not fit in the 1.7 GB of main memory allocated to each compute node used in the associated experiments.

The real-world dataset in Chapter 5 provides another insight as to why we might not represent candidate solutions as full program ASTs: half of all human-produced patches for the bugs in that dataset touched 25 lines of code or less. This is consistent with trends seen in industrial practice: most human-generated bug fixes are fairly short [141]. Automatically-generated patches are likely to also be fairly short, and two independent candidate solutions for a particular bugs might differ by only 2×25 lines of code. All other AST nodes and source code lines would be in common. Representing intermediate variants as full program source trees results in high redundancy of lines that are untouched by the actual edits, which is very inefficient.

As a result, GenProg represents each candidate in the space as a variable-length sequence of tree-based edits to the abstract syntax tree of the program. An *edit* is a modification to a subtree of the abstract syntax tree, such as "Delete the subtree rooted at this node"; we describe the specific edits the GenProg uses in generating patches in Section 3.4.3. In general, however, the tree-based edits operate over AST nodes indexed by a unique identifying number, e.g., "Delete 7" corresponds to "Delete the subtree rooted at the node uniquely internally identified by the number 7." Essentially, these edits form a small alphabet of possible mutations that are then indexed by AST node numbers. Conceptually, using a list of edits to represent variants efficiently when most programs share all but a few lines is akin to using an efficient sparse matrix representation when all but a few entries are zero.

To enable this representation, in the program parsing stage, GenProg assigns a unique ID to each statement in the AST; the edit operations refer to and operate on these IDs. Figure 3.4 illustrates this concept: the nodes corresponding to statements are assigned integer identifiers via a breadth-first traversal on the graph. The number order does not matter, so long as they uniquely identify the statements in question. Manipulating a statement (deleting or moving it, for example) also manipulates the expressions that are subtrees of that statement, as indicated by the figure.

New variants are created by adding new random edits to an existing patch, where the existing patch might be empty. The search through the space of patches progresses by composing small edits together, and recombining different sets of small edits, into larger sets comprising a patch. Variants are evaluated by applying the edit operations to the input AST, reifying the result as source code, and compiling and executing that code (Section 3.6). While tree-based edit lists may be difficult for humans to parse as they are represented internally, as they lack context about where and what is being edited, they are easy to reify as changes to source code in a well-accepted format (e.g., using `diff`, a standard tool for code differencing). Defect reports associated with patches in this type of format are more likely to be addressed by human developers, even when the patches do not correspond to the fix that the humans actually deploy [145].

The idea of exploring patches to a program instead of as candidate fixed programs has been further developed by other researchers [164], though those patches are represented as fixed-length arrays that encode all possible changes to a given program's AST, with bits in each index of the array signifying whether the change is incorporated in a particular patch. The size of the representation grows considerably with the size of the program, losing the scalability benefits we see with variable-length individuals of edits to the underlying AST.

3.4.2 Selection and population management

The previous subsection described how GenProg represents individual candidate solutions. Genetic algorithms iterate over populations of such individuals. Thus, we propose a method to generate an initial population as a starting point for a search. Line 1 of Figure 3.3 generates a random set of candidate patches to serve as this initial *population*, of size *PopSize*. This initial population is generated by creating *PopSize* random 1-edit patches using the mutation operator (see Section 3.4.3). While the number of edits in the initial individuals is tunable, one or two edits per initial candidate is typically preferred. We expect most final repairs to be small, matching the distribution of human bug fixes (short, with a very long tail) [141]. We set the size of the initial candidate patches to match our intuition that since most useful real-world patches are short, automatically-generated patches may be short as well.

The initial population Pop serves as the starting point for the iterative algorithm; new generations of candidate solutions are created from this starting point. Line 3 of Figure 3.3 implements the process by which GenProg *selects* individual variants from one population (with replacement) to copy over to the next generation. The intuition underlying selection is that, as in biology, the most fit or desirable individuals are more likely to "survive" and "reproduce" to the next generation. Less fit individuals *may* survive and reproduce, the probability that they do so is just lower.

GenProg performs selection via *tournament selection* [196]. In tournament selection, groups of k individuals are chosen at random from the population, and the individual in that random group with the highest fitness is selected. Tournaments are run until the desired population size is attained. Selection pressure can be modified by changing the tournament size k. Smaller k increases the probability that weaker variants are maintained in the population, which we

do because the search may benefit from intermediate steps that temporarily perform worse with respect to the objective function (e.g., intuitively, to escape from a local maximum, one must first walk down). We thus keep k low to help avoid local optima. We have also experimented with *stochastic universal sampling* [197], in which each individual's probability of selection is directly proportional to its fitness, and found that its results are comparable.

The selected individuals are mutated and recombined to complete their contribution to the next generation. The operators used in this phase are described in the next subsection.

3.4.3 Genetic operators

Two GP operators, mutation and crossover, create new variants from the mating pool. In the algorithm in Figure 3.3, these operators correspond to the function calls `mutate` and `crossover`. The operators take as input either an existing individual, producing one new candidate that is a modification of the original (via mutation); or two existing individuals that are recombined to produce two new candidates (via crossover). These operators are inspired by the random mutations that apply to DNA during biological evolution, and are the primary mechanism by which new candidate solutions are created from an existing population. Good GP operators should be flexible, encoding an expressive set of possible edits to the underlying genome of the candidate solutions. They should ideally produce valid genomes that correspond, in our case, to valid candidate solutions. They should also introduce sufficient variety so as to enable an effective search of the underlying space.

Because our candidates are represented as sequences of edits, mutation and recombination of the individual variants themselves is distinct from the process of evaluating the patches by applying them to the input program. This procedure, while discussed below, is most important in fitness evaluation, discussed in Section 3.6.

Crossover. Our *crossover* operator combines two individuals to produce two new "offspring" individuals. It is useful because it can combine partial solutions encompassed in existing candidates, helping the search avoid local optima. Intuitively, crossover combines the "first part" of one variant with the "second part" of another, creating offspring variants that incorporate information from two parents. Every surviving variant in a population undergoes crossover i.e,. our *crossover rate* is 1.0, though a variant will only be the parent in one such crossover per generation. *One-point crossover* [198] selects crossover points p_n and q_m in parents p and q. The first half of p (i.e., $p_1 \ldots p_n$, all edits up to and including edit p_n) is prepended to the second half of q (i.e., $q_{m+1} \ldots q_{max}$, all edits after edit q_m), and vice versa, to create two offspring.

There exist other crossover options, with tradeoffs in terms of the observed behavior combinations that result. For the purposes of controlled experimentation, to isolate certain aspects of our algorithm, we also define a *patch subset* crossover operator as a variation of the well-known *uniform* crossover operator [199] tailored for the program repair domain. It takes as input two parents p and q represented as ordered lists of edits (Section 3.4.1). The first (resp. second)

Input: Candidate patch C to be mutated.
Input: Subject program P
Output: Mutated candidate C'.
1: **let** $stmt_d = \mathsf{choose}(P)$
2: **let** $op = \mathsf{choose}(\{\mathsf{insert}, \mathsf{replace}, \mathsf{delete}\})$
3: **if** $op = \mathsf{insert} \vee op = \mathsf{replace}$ **then**
4: **let** $stmt_s = \mathsf{choose}(P)$ **return** $\{C, op(stmt_d, stmt_s)\}$
5: **else if** $op = \mathsf{delete}$ **then return** $\{C, \mathsf{delete}(stmt_d)\}$
6: **end if**

Figure 3.6: The mutation operator. The way statements are selected by choose() to serve as sites for mutations or as sources for the insertion and replacement operators is discussed in Section 3.5.

offspring is created by appending p to q (resp. q to p) and then removing each element with independent probability one-half.

We compare the utility of these different crossover approaches in Chapter 6. Unless otherwise noted, however, other experiments in this dissertation use one-point crossover.

Mutation Our *mutation* operator randomly modifies an individual candidate to produce a new candidate solution. Mutation modifies an individual by adding a single new tree-based edit to the list of edits that comprises the patch. Because our primitive unit is the statement, we require mutation operators that can flexibly express a variety of changes at that level of granularity. This can be contrasted with the mutation operators that are common in standard genetic algorithm applications, which typically involve single bit flips or simple symbolic substitutions. Our mutation operator consists of either *delete* (the entire statement is deleted), *insert* (another statement is inserted after it), or *replace* (one statement is replaced by another statement; equivalent to a delete followed by an insert to the same location). GenProg typically chooses from these options with uniform random probability, though we explore improvements on this model in Chapter 6.

Figure 3.6 shows the high-level pseudocode for mutation. To add a single mutation to a candidate patch, a destination statement $stmt_d$ is chosen from the program as the location of the change (line 1). GenProg then selects one of the three mutation operators to apply at $stmt_d$, with equiprobability (line 2). If the mutation operator selected is an insertion or a replacement, GenProg must selects a source statement $stmt_s$ (line 4) to insert or replace at $stmt_d$.

Evaluating a candidate patch involves applying these edits in order to the input program to produce a changed program. This procedure happens at fitness evaluation time (Section 3.6), and therefore is not shown in Figure 3.6. When each edit is applied, GenProg either deletes $stmt_d$ by replacing it with the empty block, inserts statement $stmt_s$ after $stmt_d$ by replacing $stmt_d$ with a new block that contains $\{stmt_d, stmt_s\}$ or replaces $stmt_d$ with another statement $stmt_s$. Source statements are not modified when inserted, although we note that intermediate variants may fail to compile if code is inserted which references out-of-scope variables. Because deletions transform $stmt_d$ into an empty block statement, a deleted statement may be modified in a later mutation operation.

Referring back to the worked example: one random, 1-edit candidate solution is the patch ["Delete(6)"]. When applied to the original AST in Figure 3.4, this candidate produces the modified AST in Figure 3.5. Note that because deletions and insertions modify in such a way that the code maintains the high-level tree structure (replacing an existing statement with the empty block for a deletion, for example), the original numbering scheme produced in the pre-processing step is maintained as an invariant over the course of mutation, and earlier mutations in the genome do not invalidate the application of later mutations (though they may render them less impactful).

The human-produced patch for the same defect, shown in Figure 3.1(b) is most simply expressed as an **insert** operation at the start of the `block` that corresponds to the `then` branch of the `if` condition on line 13. The candidate code to be inserted comes from elsewhere in the same program (see Section 3.5) and thus can be identified uniquely via the same preprocessing numbering step described above, though the inserted code is not shown in the figure.

Even though the types of mutations are restricted and their impact is limited to the statement level of the abstract syntax tree, the space of changes that can be encoded by the edits we have defined is very large. We have omitted so far considerations about how statements are selected as the destination statement $stmt_d$ as well as how source statements $stmt_s$ are selected for the replacement and insertion operators. These concerns are critically related to both the definition of the search space and the scalability of the search overall; we address them in depth in the next section.

3.5 Search space and localization

We have thus far presented an outline for a genetic programming search for a patch that fixes a bug in a program, and described the components of the algorithm designed to efficiently, scalably, and expressively traverse the search space. However, that space is infinite and extremely complex, as evidenced by the complexity of the human debugging problem. Fortunately, there are additional domain-specific insights that can be brought to bear on traversing it intelligently.

Consider again the bug in `nullhttpd` (Figure 3.1(a)), and its abstract syntax tree (Figure 3.4). This code snippet represents only a small portion of the 5575-line program; we have localized it for the purposes of presentation. Displaying all 5575 lines is unnecessary, however, because *not all program locations are equally likely to be good choices for changes to fix the bug.* Rather, most bug-fixing changes are small, and much of the human debugging process, and considerable software engineering research, is devoted to localizing a bug to a small code region. In GenProg's terms, this means we can localize a bug to reduce the number of destination statements $stmt_d$ that can be selected as locations for mutation.

Additionally, once a mutation has been selected, consider the source statement $stmt_s$ that is used as insertion or replacement code. Where should these source statements come from? Other work in GP has used, for example, grammar-based approaches, which define rules that are then used to randomly generate valid code [166].

However, a grammar-based approach for syntactically valid C results in a tremendous number of options for any particular $stmt_s$. Instead, we observe that *a program that makes a mistake in one location often handles a similar situation correctly in another* [48]. As a result, GenProg restricts source statements $stmt_s$ to code found elsewhere in the same program. This approach applies to **nullhttpd**, because although the POST request handling in **ProcessRequest** does not do a bounds check on the user-specified content length, the **cgi_main** function, implemented elsewhere, does:

```
502  if (length <= 0) return null;
```

This code can be copied and inserted into the buggy region, which is what is shown in the fixed version of the program (Figure 3.1(b)).

The search space for program repair is therefore defined by the locations that can be changed, the mutations that can be applied at each location, and the statements that can serve as sources of the repair. We constrain the search via probability distributions defined over the space, e.g., the probability that a given location is selected as $stmt_d$, a given mutation is applied, or a given source statement $stmt_d$ is selected. These choices define the possible patches that the search may enumerate, which in turn relates to the scalability of the search. We formally parametrize these restrictions along three dimensions: fault space, mutation space, and fix space.

3.5.1 Fault space

Each mutation step begins by selecting a destination statement $stmt_d$ to which to apply a random mutation. In Figure 3.6, this selection is performed by the function choose(), applied to the statements in the program P. Allowing the mutation operators to randomly change any statement in P is inefficient, and GenProg instead restricts mutations to program locations associated with incorrect behavior. These locations (i.e., program statements) are weighted to influence their probability of being mutated, such as by existing fault localization techniques (e.g. Tarantula [143] or suspiciousness values [161]) or other, similar heuristics.

GenProg focuses repair efforts on statements that are visited by the negative test cases, excluding code that such tests do not execute. To initialize this weighting, when GenProg assigns each statement a unique number in the preprocessing phase, it also inserts code to log an event (visit) each time the statement is executed. Duplicate statements are removed from the list: that is, we do not assume that a statement visited frequently (e.g., in a loop) is likely to be a good repair site. Any statement visited during the execution of a negative test case is a candidate for repair, and its initial weight is set to 1.0. All other statements are assigned a weight of 0.0 and never modified.

These weights may be additionally tuned. An initial intuition is that mutation should focus as much as possible on portions of the source code that are strongly correlated with the failing test case. As such, it is possible to modify the initial weighting by observing which statements executed by the negative test case are also executed by positive test cases, and set the weights of such statements differently. We call this weight W_{Path}, and it serves as a tunable

parameter of the algorithm. In many of the experiments in this dissertation, we set W_{Path} to 0.1, to favor mutations to regions of the source code strongly associated with the failing test case (which are weighted 10x as highly), without removing other portions of the source code entirely from consideration. We investigate this intuition and its implications in more detail in Chapter 6.

To illustrate using the code shown in Figure 3.1, by instrumenting the program and running it on the test cases in a preprocessing step, we record which statements are executed on which test cases. The standard regression test visits lines 1–12 and 18 (and lines in `ProcessGETRequest`). The test case demonstrating the error visits lines 1–11 and 13–18. Mutation and crossover therefore focus on lines 13–17, which exclusively implement POST functionality, and are most strongly related to the defect in question. Those lines also contain the site of the human repair, suggesting that our intuition matches practice for this particular bug.

Phrased more formally, and in terms of the pseudocode in Figure 3.3: for a given program, defect, set of tests T, test evaluation function $Pass : T \to \mathbb{B}$, and set of statements visited when evaluating a test $Visited : T \to \mathcal{P}(Stmt)$, we define the fault localization function $faultloc : Stmt \to \mathbb{R}$ to be:

$$faultloc(s) = \begin{cases} 0 & \forall t \in T.\ s \notin Visited(t) \\ 1.0 & \forall t \in T.\ s \in Visited(t) \implies \neg Pass(t) \\ W_{Path} & \text{otherwise} \end{cases}$$

That is, a statement never visited by any test case has zero weight, a statement visited only on a bug-inducing test case has high (1.0) weight, and statements covered by both bug-inducing and normal tests have moderate (W_{Path}) weights. Other fault localization schemes could potentially be plugged directly into GenProg [200].

We define the size of the fault space as the sum of all weights over all program statements.

3.5.2 Mutation space

The search space is further constrained by the set of mutations that are possible at each location and their selection probability. While the mutations are designed to be general (see Section 3.4.3), not all mutation choices may apply at a given statement in a particular location. For example, given that candidate source statements are restricted to the code found elsewhere in the same program, there may not be any statements that do not reference out-of-scope variables at a given location $stmt_d$. Such an insertion would produce a program that does not compile, which we seek to avoid (see Section 3.5.3). Typically, however, available mutations are selected with equal random probability, though we investigate this heuristic in Chapter 6.

3.5.3 Fix space

The insertion and replacement operators require a piece of code, $stmt_s$ to be inserted or serve as the replacement. We refer to the set of candidate pieces of movable code, from which new repairs are constructed, as the *fix space*. Ideally, the code considered as part of the fix space has a high probability of serving as useful sources for a repair in the program in question. On the other hand, too many options will lead to an intractably large search space.

To balance these concerns, we begin by restricting candidate statements s to program statements that *appear in the same program that GenProg is trying to repair.* This code can come from anywhere in the program, not just along the execution path associated with the failing test case, and a statement's fault space weight does not influence the probability that it is selected as a candidate repair. This design choice reflects our intuition about related code and changes, about programmer knowledge encoded elsewhere in the program, and about the fact that programmers program correctly most of the time [48]. Intuitively, a bug in a webserver will be fixed by code that has something to do with a webserver; restricting candidate fixes to within the same program provides, by definition, a realistic domain restriction on the type of code that will be used as a candidate fix. However, correct code is not found necessarily on the failing execution path, and so the probability that a given statement is selected as a source for a candidate repair operation is unrelated to its execution on a failing test case.

We demonstrate this intuition using our running example in the introduction to this section. Restricting attention to the code within `nullhttpd` provides sufficient options for repairing the bug, because of the nature of software and the general competence of programmers, while simultaneously reducing the number of candidate pieces of source code $stmt_s$ to a smaller, domain-appropriate set.

We introduce the term *fix localization* to refer to the problem of identifying a suitable *source* of insertion/replacement code from the fix space (analogous to using fault localization to identify suitable locations to change to address a fault). We explore ways to improve fix localization beyond blind random choice. As a start, we restrict inserted code to that which includes variables that are in-scope at the destination (so the result compiles) and that are visited by at least one test case (because we hypothesize that certain common behavior may be correct). We refer to this restriction as a *semantic check*. For a given program and destination we define the function $\mathit{fixloc} : \mathit{Stmt} \rightarrow \mathcal{P}(\mathit{Stmt})$ as follows:

$$\mathit{fixloc}(d) = \left\{ s \;\middle|\; \exists t \in T.\ s \in \mathit{Visited}(t) \ \wedge\ \mathit{VarsUsed}(s) \subseteq \mathit{InScope}(d) \right\}$$

The semantic check is a special case of the operators proposed by Orlov and Sipper for well-typed Java bytecode mutation [162], applied to weakly typed C programs. Fix localization is context-sensitive [168]: given a destination statement $stmt_d$, we only consider bringing in a statement $stmt_s$ if the variables $stmt_s$ references are in scope at $stmt_d$.

3.6 How are individuals evaluated for desirability?

The final internal component of the GP algorithm that we have yet to define is the fitness function, which evaluates the acceptability of a candidate solution and should measure or approximate its distance from the desired solution. It serves as the objective function, providing the termination criterion for the search and guiding the selection of variants between intermediate generations.

In our application, the fitness function should approximate the distance of a candidate patch from a patch that creates a program with all of the desired behavior encoded by its test cases, both positive and negative. The test cases encode requirements; the fitness function thus measures how many of those requirements an intermediate program meets.

GenProg thus measures fitness as a weighted sum of the number of test cases passed by the program produced by applying a candidate patch to the input program. The tree-based edits comprising an individual patch are applied to the input AST in order, the resulting AST is reified to source code and compiled to an executable, and the executable is run on the test cases. GenProg records which test cases pass, and the final fitness of the variant is their weighted sum. A variant that does not compile has fitness zero.

The weighting scheme for passed test cases is set heuristically, with the tunable parameters W_{PosT} serving to weight the passing positive test cases, and W_{NegT} weighting the negative test cases:

$$\begin{aligned} \mathsf{fitness}(C) &= W_{PosT} \times |\{t \in PosT \mid P' \text{ passes } t\}| \\ &+ W_{NegT} \times |\{t \in NegT \mid P' \text{ passes } t\}| \end{aligned}$$

Where P' is the program that results from applying candidate patch C to input program P. We typically set W_{PosT} and W_{NegT} such that negative tests are weighted twice as heavily as the positive tests, which emphasizes repairing the fault while weighting the positive test cases evenly. In typical usage, there are many more positive test cases than negative tests, and thus this weighting rewards exploration in the direction of a repair for the initially-failing tests. In related work [201] we have observed that it is possible to select more precise weights, as measured by the fitness distance correlation metric [185], which approximates the precision of the fitness function relative to a known ground truth.

The "Delete(6)" patch visualized in Figure 3.5 is likely to have moderate fitness value (making several simplifying assumptions about the rest of the codebase). The candidate patch deletes the code that copies the request type from the request the header. This value is used at lines 11 and 13 to dispatch to the appropriate handling function. Without the copy at line 7, neither condition can evaluate to true, and neither GET nor POST requests will be handled. As a result, the bug-inducing POST request in the negative test case will not crash this server, but positive test cases checking GET or normal POST functionality will fail. The final resulting fitness thus depends on both which test cases pass (the negative test case, in this instance) and the weights selected for W_{PosT} and W_{NegT}.

In practice, regression test suites can be very time-consuming to run in their entirety. Fortunately, GP performs well in the face of noisy fitness functions [201], which allows us to exploit existing test suite reduction and prioritization techniques in the interest of scalability. GenProg samples the positive test cases to produce a smaller set of tests that can be run quickly. If an intermediate variant passes the entire positive sample as well as all the negative test cases, GenProg tests it against the entire set of test cases.

In terms of the algorithm in Figure 3.3, the function SampleFit applies a candidate patch to the original program and evaluates the result on a random sample of the positive tests as well as all of the negative test cases. SampleFit chooses a different test suite sample each time it is called. FullFitness evaluates to true if the candidate patch, when applied to the original program, passes all of the test cases. For efficiency, only variants that maximize SampleFit are fully tested on the entire test suite. Only repairs that maximize FullFitness are returned as initial repairs.

For full safety, the test case evaluations can be run in a virtual machine or similar sandbox with a timeout. Standard software fault isolation through address spaces may be insufficient. For example, a multithreaded program that creates new processes can evolve into a variant that overwhelms the test machine with uncontrolled thread forking. In addition, the standard regression testing practice of limiting execution time is even more important in this setting, where program variants may evolve infinite loops.

3.7 Minimization

The search terminates successfully when GenProg discovers an initial repair that causes the input program to pass all provided test cases, maximizing FullFitness as explained in the previous section. However, due to randomness in the mutation and crossover operators, the initial repair typically contains at least an order-of-magnitude more changes than are necessary to repair the program, rendering the repairs difficult to inspect for correctness.

For example, in searching for a patch that repairs the code in Figure 3.1, GenProg could identify a patch that both inserts a copy of the appropriate bounds check in the appropriate place, followed by a random insertion of `return ProcessGetRequest(socket,length)` at line 22, after the original `return`. This insertion is not dangerous, because it will never be executed, but it does not contribute to the repair. It can therefore be safely removed, as including it in the final repair is unnecessarily confusing at best and potentially harmful to future maintenance efforts.

Therefore, GenProg minimizes the initial repair to produce the final repair, expressed as a list of edits in standard Unix `diff` format. The minimization process finds a subset of the initial repair edits from which no further elements can be dropped without causing the program to fail a test case (a *1-minimal subset*). Intuitively, the minimization process considers each difference between the primary repair and the original program and discards every difference that does not affect the repair's behavior on any of the test cases; the changes that remain comprise the 1-minimal subset. GenProg uses *delta debugging* [202] to efficiently compute this subset, which is $\mathcal{O}(n^2)$ worst case [203].

Because this process is deterministic and considerably faster than the rest of the search, we can be finer-grained in computing and minimizing the differences between the original and the patched solution than GenProg is in generating the initial repair. We convert the initial repair into a set of changes that can apply to any node in the abstract syntax tree using the DIFFX XML differencing algorithm [204], adapted to work on CIL ASTs. Modified DIFFX generates a list of tree-structured edit operations similar in style to those that comprise the initial patch, but at a lower level of granularity, admitting the potential of finer-grained minimized patches.

Once the patch has been minimized, GenProg applies the minimal patch to the input program to produce a repaired program and computes the syntactic difference between them to produce the corresponding `diff` patch, which is a well-established format for reviewing bug fixes. In the results presented in this dissertation, patch sizes are reported in the number of lines of a Unix `diff` patch, not DIFFX operations.

3.8 Summary and conclusions

We have described GenProg, an algorithm that targets the automatic repair of patches for bugs in large, real-world programs. As input, GenProg takes a program with a bug, at least one negative test case that encodes the bug (and is initially failing), and at least one positive test case that encodes other required program behavior (and is initially passing). GenProg uses genetic programming to perform a stochastic search over the space of patches to the input program, with the goal of identifying a patch that, when applied, causes the program to pass all of its input test cases. We propose a novel representation, test-case-based fitness function, and set of mutation operators; a novel post-processing minimization step that builds on tree-based differencing algorithms and delta debugging to control code bloat; and a novel characterization and intelligent limitation of the search space, to construct an algorithm with an eye towards scalability, expressive power, and quality output. The guiding research and design principle remains, at a high level, to provide a technique that is applicable to the types of bugs and systems that human developers address in practice. Our goal is not to provably fix every defect of a particular type that humans encounter, but rather to provide a broadly applicable and cost-effective approach to alleviate the cost of bug repair in the wild.

In subsequent chapters, we evaluate our hypotheses about GenProg, and provide evidence to substantiate our claims about its effectiveness. We also analyze its scaling and performance behavior, and evaluate in depth several of the design choices we have presented in this chapter.

Chapter 4

GenProg is expressive, and produces high-quality patches

SECURITY vulnerabilities comprise a particularly important class of defects when it comes to program repair. They are a source of considerable real-world economic and human damage, as malicious attackers take advantage of them to sabotage running systems and compromise and take advantage of private user data [63]. They are very important to patch both quickly and correctly. When it comes to their repair, there are many different classes of security defects that have historically all been handled differently [64].

Human-generated repairs of important bugs in important programs are known to be error-prone [8]; automatic repair quality on these types of programs is a particularly compelling problem. Security vulnerabilities therefore provide excellent case studies for automatic program repair: They exist in wide variety, are independently important to fix, and are useful in illustrating the repair process in the context of large programs with publicly documented bugs. They also provide a natural framework for a quantitative study of the quality of an automatically-generated patch.

As a result, we begin our evaluation of GenProg with respect to the properties necessary in a viable real-world automated repair solution (see Chapter 1) by automatically repairing real-world security vulnerabilities. These experiments examine the first component of GenProg's expressive power as well as the quality of the patches it produces. We present evidence to substantiate our claims that GenProg can repair a variety of real bugs in a variety of real programs with little *a priori* knowledge; and that it produces high quality repairs, in that they address the underlying defect, maintain required functionality, and avoid introducing new defects. Expressive power is important for automatic program repair because it is a key benefit of the current state of the art in program repair: human developers. Human ingenuity allows developers to deal with different types of bugs. Repair quality is important because automatic repair is only useful if humans trust its results enough to deploy them.

In Chapter 1, we formalized these claims with the following falsifiable hypotheses:

> **Hypothesis 1:** Without defect- or program-specific information, GenProg can repair at least 5 different defect types and can repair defects in at least least 10 different program types.

and

> **Hypothesis 4:** GenProg-produced patches maintain existing program functionality as measured by existing test cases and held-out indicative workloads; they do not introduce new vulnerabilities as measured by black-box fuzz testing techniques; and they address the underlying cause of a vulnerability, and not just its symptom, as measured against exploit variants generated by fuzzing techniques.

In this chapter, we evaluate these hypotheses by presenting repair results on a benchmark set designed to include a variety of defect types in a variety of programs. We describe several of those repairs in detail to provide a qualitative discussion of the repairs GenProg has found. The case studies, as well as many of the errors in the benchmark set, are associated with publicly-reported security vulnerabilities, because they provide particularly compelling case studies to evaluate automatic software repair.

We leave to later chapters the question of whether GenProg is human competitive in terms of time and economic cost (and explicitly its scalability) and in terms of the proportion of bugs in software engineering practice that GenProg can fix. We divide the presentation this way because of the types of experiments involved in evaluating the respective claims. The experimental framework developed in this chapter is designed to enable an examination of GenProg's effect on particular bugs; the framework in Chapter 5 focuses on designing and building a large set of defects that are indicative of those that human developers fixed over a period of time, without regard for bug type. In other words, this chapter provides an existence proof of GenProg's ability to repair different types of *particular* bugs, while Chapter 5 provides a survey of GenProg's effectiveness on a broad array of them.

The structure of this chapter follows its contributions, which are:

- A novel benchmark set of bugs and programs designed to evaluate expressive power, comprised of 16 programs spanning multiple domains and totaling 1.25M lines of C code. Each program is associated with a defect, and the defects span eight types (Section 4.1.1): infinite loop, segmentation fault, remote heap buffer overflow to inject code, remote heap buffer overflow to overwrite variables, non-overflow denial of service, local stack buffer overflow, integer overflow, and format string vulnerability. The benchmark programs include Unix utilities, servers, media players, text processing programs, and games. Many of the bugs are taken from publicly reported security vulnerabilities.

- Experimental evidence to evaluate our first hypothesis and demonstrate that GenProg is expressive, in that it can generate repairs to eight different vulnerability types in a variety of different programs without *a priori* specification of bug or program type (Section 4.2).

- A manual, qualitative description of a set of case study bug repairs generated by GenProg for publicly-available security vulnerabilities (Section 4.3)

- A quantitative analysis of repair quality (Section 4.4), including novel experimental procedures for evaluating automatic repair in the context of a closed-loop system for detection and repair of security vulnerabilities. We use indicative workloads, fuzz testing, and variant bug-inducing input, and our findings suggest that the repairs are not fragile memorizations of the input, but instead address the underlying defects while retaining required functionality.

4.1 Expressive power experimental setup

Can GenProg repair at least 5 different defect types in at least least 10 different types of programs? This question motivates a benchmark set comprised of a variety of bugs in a variety of programs. We use GenProg to repair each of those bugs in turn and measure GP success rate—that is, the percentage of random trials that succeed—and average repair time. We measure time both by wall-clock time and by the number of fitness evaluations to a repair, a hardware- and program-independent measure of search effort.

Finding patches that cause test cases to pass is half the battle; the other half is substantiating the claim that those patches are acceptable. In industrial practice, a number of different techniques are used to validate candidate bug fixes. In the absence of a human developer to inspect the patches, we require additional evidence to be confident that these automatically-generated patches are acceptable. As a result, we use the patches GenProg finds for the bugs studied in this chapter to characterize the types of patches GenProg generates and what types of changes they affect in practice. We look specifically at how and whether the patches insert or delete code, and what effect these changes have on the bugs in question. We also describe several patches in detail to guide a qualitative discussion of their acceptability. Finally, we quantitatively evaluate the quality several of these patches using a novel experimental framework; we explain how in Section 4.4.

Program	Lines of Code	Positive Tests	Program Description	Fault
`gcd`	22	5x human	example	infinite loop
`zune`	28	6x human	example [66]	infinite loop†[1]
`uniq utx 4.3`	1146	5x fuzz	duplicate text processing	segmentation fault
`look utx 4.3`	1169	5x fuzz	dictionary lookup	segmentation fault
`look svr 4.0 1.1`	1363	5x fuzz	dictionary lookup	infinite loop
`units svr 4.0 1.1`	1504	5x human	metric conversion	segmentation fault
`deroff utx 4.3`	2236	5x fuzz	document processing	segmentation fault
`nullhttpd 0.5.0`	5575	6x human	webserver	remote heap buffer overflow (code)†[2]
`openldap 2.2.4`	292598	40x human	directory protocol	non-overflow denial of service†[3]
`ccrypt 1.2`	7515	6x human	encryption utility	segmentation fault†[4]
`indent 1.9.1`	9906	5x fuzz	source code processing	infinite loop
`lighttpd 1.4.17`	51895	3x human	webserver	remote heap buffer overflow (variables)†[5]
`flex 2.5.4a`	18775	5x fuzz	lexical analyzer generator	segmentation fault
`atris 1.0.6`	21553	2x human	graphical tetris game	local stack buffer exploit†[6]
`php 4.4.5`	764489	3x human	scripting language	integer overflow†[7]
`wu-ftpd 2.6.0`	67029	5x human	FTP server	format string vulnerability†[8]
total	1246803			

Table 4.1: Open-source benchmark programs and associated defects used in the experiments in this chapter. All programs are written in the C programming language, and the size of the program is given in lines of code (LOC). A † indicates an openly-available exploit with an associated defect report.

4.1.1 Benchmarks

Programs. Finding good benchmark sets of bugs in programs is one of the dominant experimental challenges throughout this dissertation. Good benchmarks are critical to high-quality empirical science: "Since benchmarks drive computer science research and industry product development, which ones we use and how we evaluate them are key questions for the community." [205] A good benchmark defect set should be *indicative* and *generalizable*, and should therefore be drawn from programs representative of real-world systems. The defects should illustrate real bugs that human developers would consider important, and be easy to reproduce.

Existing benchmark suites that are used in other software engineering research evaluations, such as SPEC or Siemens [206] do not fulfill these requirements. The SPEC programs were designed for performance benchmarking

[1] http://pastie.org/349916 (Jan. 2009)
[2] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2002-1496
http://www.securiteam.com/exploits/5VP0P0U8AQ.html
[3] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-2952
http://www.securityfocus.com/bid/30013/exploit
[4] http://sourceforge.net/tracker/index.php?func=detail&aid=598800&group_id=40913&atid=429289
[5] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-4727
http://www.milw0rm.com/exploits/4437
[6] http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=290230
[7] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-1885
http://www.php-security.org/MOPB/MOPB-39-2007.html
[8] http://www.cert.org/advisories/CA-2000-13.html
http://www.securityfocus.com/bid/1387/exploit — wuftpd-god.c

and do not contain the intentional semantic defects that are required for the automated repair problem. The Siemens suite does provide programs with test suites and faults. However, it was designed for controlled evaluation of software testing techniques. As a result, the test suites maximize statement coverage, the faults are almost exclusively seeded, and the programs are fairly small. The benchmarks are thus not indicative of the large software systems that exist in modern practice.

Because existing benchmark sets are unsuitable for our experimental goals, we identify a new set of reproducible defects in open-source programs. The approach we take in this chapter is to individually identify bugs "in the wild" through case studies, careful search through bug databases, industrial partnerships, and word-of-mouth (this approach has been followed by other researchers as well [13, 200]). Our goal is to identify as broad a range of defects in as many different types of programs as possible.[1]

The resulting benchmark programs, and associated defects, are shown in Table 4.1. `gcd` is a small example based on Euclid's algorithm for computing greatest common divisors. `zune` is a fragment of code that caused all Microsoft Zune media players to freeze on December 31st, 2008; it is an increasingly-used example in other work on debugging and program repair [207]. The Unix utility programs are open source, and their faults were taken from Miller *et al.*'s work on *fuzz testing*, in which programs crash when given random inputs [73]. The remaining benchmarks are taken from public vulnerability reports that correspond to security vulnerabilities that developers have addressed in the field. These programs total 1.25 LOC and the repaired errors span 8 defect classes (infinite loop, segmentation fault, remote heap buffer overflow to inject code, remote heap buffer overflow to overwrite variables, non-overflow denial of service, local stack buffer overflow, integer overflow, and format string vulnerability).

Test Cases. GenProg requires as input both the program source code and a set of test cases that demonstrate the bug under repair and other required functionality, as described in Section 3.6). As we show in Chapter 6, test case execution time for intermediate variants dominates GenProg execution time. This is why we take advantage of the GA's tolerance of noisy fitness function and use both a sampling fitness function and a full fitness function, to reduce the number of test case executions required in intermediate stages.

The benchmarks in this chapter, however, are intended to evaluate claims of expressive power; we are less concerned about scalability to indicative test suites. We therefore designed the scenarios to admit deterministic reproducibility of the bug in question, with an eye towards variety, ignoring for now issues of test suites of indicative size. For the purposes of simplicity, we keep the test suites small, with an eye towards creating a small set of test cases that exercises indicative positive functionality. We do not sample the resulting fitness function.

For each of the programs in Table 4.1, we developed a single negative test case that elicits the given fault. For the Unix utilities, we selected the first fuzz input that evinced a fault; for the others, we constructed test cases based on the vulnerability reports (Section 4.3 gives concrete examples that illustrate this procedure). We selected a small number

[1] We propose another technique for developing benchmarks to evaluate a different set of claims about GenProg's effectiveness in Chapter 5.

(e.g., 2–6) of positive test cases per program. In some cases, we used non-crashing fuzz inputs; in others, we manually created simple cases, focusing on testing relevant program functionality; for **openldap**, we used part of its provided test suite. When evaluating GenProg's scalability (Chapter 5) we use off-the-shelf developer-written test cases. For this chapter, focusing on a small set of indicative test cases allows us to separate scalability concerns (that is, the effect of test suite size on GenProg run-time) from quality concerns (that is, the dangers posed by inadequate test suites, and the quality of the resulting repairs).

4.1.2 Experimental parameters

The experiments in this chapter were conducted on a quad-core 3 GHz machine.

We report results for one set of global GenProg parameters that work well in practice and that are consistent with the approach described in the previous chapter. We chose pop_size $= 40$, which is small compared to typical GP applications. We stop an individual trial if an initial repair is found; otherwise, the GP is run for ten generations (also a small number). Because the test suite sizes are small by design, we set the sample rate in SampleFit to 1.0, and evaluate each intermediate candidate on the entire test suite. For fitness computation, we set $W_{PosT} = 1$ and $W_{NegT} = 10$, weighting the negative test cases approximately twice as highly as the positive test cases (as there are typically 5-10 test cases associated with these benchmarks).

We perform 100 random trials for each program and report the percentage of trials that produce a repair; average time to the initial repair in a successful trial; and time to minimize a final repair. The *fault space size* is the weighted sum of statements on the negative path and provides a partial estimate of the complexity of the search space. Statements that appear only on the negative path receive a weight of 1.0, while those also on a positive path receive a weight of $W_{Path} = 0.1$. Additional experiments show that GenProg is robust to changes in many of these parameters, such as population size. Representation and operator choices do impact repair success, however, as we explore in Chapter 6.

Given the same random seed, each trial is deterministically reproducible[2] and, if successful, leads to the same initial repair. However, with unique seeds and for some programs, GenProg generates several different patches over many random trials. For example, over 100 random trials, GenProg produces several different acceptable patches for **ccrypt**, but only ever produces one such patch for **openldap**. Such differences depend on the program, error, and patch type. We thus report average patch lengths when appropriate. We evaluate the implications of a search for multiple different repairs to the same bug in Chapter 5. We also report the time to minimize the initial repairs to produce the final repairs, as well as the size of each type of repair, to demonstrate the effect of minimization both on the repair itself and on the run time of the algorithm.

[2]Recall that GenProg assumes the test cases are deterministic.

Program	Fault space	Time	Fitness	Success	Size	Time	Fitness	Size	Effect
`gcd`	1.3	153 s	45.0	54%	21	4 s	4	2	Insert
`zune`	2.9	42 s	203.5	72%	11	1 s	2	3	Insert
`uniq`	81.5	34 s	15.5	100%	24	2 s	6	4	Delete
`look-utx`	213.0	45 s	20.1	99%	24	3 s	10	11	Insert
`look-svr`	32.4	55 s	13.5	100%	21	4 s	5	3	Insert
`units`	2159.7	109 s	61.7	7%	23	2 s	6	4	Insert
`deroff`	251.4	131 s	28.6	97%	61	2 s	7	3	Delete
`nullhttpd`	768.5	578 s	95.1	36%	71	76 s	16	5	Both
`openldap`	25.4	665 s	10.6	100%	73	549 s	10	16	Delete
`ccrypt`	18.01	330 s	32.3	100%	34	13 s	10	14	Insert
`indent`	1435.9	546 s	108.6	7%	221	13 s	13	2	Insert
`lighttpd`	135.8	394 s	28.8	100%	214	139 s	14	3	Delete
`flex`	3836.6	230 s	39.4	5%	52	7 s	6	3	Delete
`atris`	34.0	80 s	20.2	82%	19	11 s	7	3	Delete
`php`	30.9	56 s	15.5	100%	139	94 s	11	10	Delete
`wu-ftpd`	149.0	2256 s	48.5	75%	64	300 s	6	5	Both
average	573.52	356.5 33.63	77.0%	67.0	76.3 s	8.23	5.7		

Table 4.2: Experimental results on 1.25M lines of source code from 16. We report averages for 100 random trials. The 'Positive Tests' column describes the positive tests. The Fault space size column gives the sum of the weights of all statements in the program. 'Initial Repair' gives the average performance for one trial, in terms of 'Time' (the average time taken for each successful trial), 'Fitness' (the average number of fitness evaluations in a successful trial), 'Success' (how many of the random trials resulted in a repair). 'Size' reports the average Unix `diff` size between the original source and the primary repair, in lines. 'Final Repair' reports the same information for the production of a 1-minimal repair from the first initial repair found; the minimization process always succeeds. 'Effect' describes the operations performed by an indicative final patch: a patch may insert code, delete code, or both insert and delete code.

When calculating fitness, we memoize fitness results based on the pretty-printed abstract syntax tree so that two variants with different ASTs but identical source code are not evaluated twice. Similarly, variants that are copied unchanged to the next generation are not reevaluated. Beyond such caching, no additional optimizations are considered for these experiments.

4.2 Repair Results

Figure 4.2 summarizes repair results for the sixteen C programs in Table 4.1. The 'Initial Repair' heading reports timing information for the GP phase and does not include the time for repair minimization. The 'Time' column reports the average wall-clock time per trial that produced a primary repair; execution time is analyzed in more detail in Chapter 6. Repairs are found in 357 seconds on average. The 'fitness' column shows the average number of fitness evaluations per successful trial, which we include because fitness function evaluation is the dominant expense in most GP applications and the measure is independent of specific hardware configuration. The 'Success' column gives the fraction of trials that were successful. On average, over 77% of the trials produced a repair, although most of the benchmarks either

succeeded very frequently or very rarely. Low success rates can be mitigated by running multiple independent trials in parallel. The 'Size' column lists the size of a `diff` of the primary repair in lines.

The 'Final Repair' heading gives performance information for transforming the primary repair into the final repair and a summary of the effect of the final repair, as judged by manual inspection. Minimization is deterministic and takes less time and fewer fitness evaluations than the initial repair process. The final minimized patch is quite manageable, averaging 5.1 lines.

Of the 16 patches, seven insert code (`gcd`, `zune`, `look-utx`, `look-svr`, `units`, `ccrypt`, and `indent`) seven delete code (`uniq`, `deroff`, `openldap`, `lighttpd`, `flex`, `atris`, and `php`), and two both insert and delete code (`nullhttpd` and `wu-ftpd`). Note that this does not speak to the sequence of mutations that lead to a given repair, only the operations in the final patch.

Manual inspection suggests that the produced patches are acceptable; we describe specific patches in Section 4.3. We note that patches that delete code do not necessarily degrade functionality because the deleted code may have been included erroneously, or the patch may compensate for the deletion with an insertion. The `uniq`, `deroff`, and `flex` patches delete erroneous code and do not degrade untested functionality. The `openldap` patch removes unnecessary faulty code (handling of multi-byte BER tags, when only 30 tags are used), and thus does not degrade functionality in practice. The `nullhttpd` and `wu-ftpd` patches delete faulty code and replace them by inserting non-faulty code found elsewhere. The `wu-ftpd` patch disables verbose logging output in one source location, but does not modify the functionality of the program itself, and the `nullhttpd` patch does not degrade functionality. The effect of the `lighttpd` patch is machine-specific: it may reduce functionality on very long messages, though in our experiments, it did not. More detailed patch descriptions are provided in Section 4.3; we evaluate repair quality using indicative workloads and held-out fuzz testing in Section 4.4.

In many cases it is also possible to insert code without negatively affecting the functionality of a benchmark program. The `zune` and `gcd` benchmarks both contain infinite loops: `zune` when calculating dates involving leap years, and `gcd` if one argument is zero. In both cases, the repair involves inserting additional code: For `gcd`, the repair inserts code that returns early (skipping the infinite loop) if one argument is zero. In `zune`, code is added to one of three branches that decrements the day in the main body of the loop (allowing leap years with exactly 366 days remaining to be processed correctly). In both of these cases, the insertions are such that they apply only to relevant inputs (i.e., zero-valued arguments or leap years), which explains why the inserted code does not negatively impact other functionality. Similar behavior is seen for `look-svr`, where a buggy binary search over a dictionary never terminates if the input dictionary is not pre-sorted. Our repair inserts a new exit condition to the loop (i.e., a guarded `break`). A more complicated example is `units`, in which user input is read into a static buffer without bounds checks, a pointer to the result is passed to a `lookup()` function, and the result of `lookup()` is possibly dereferenced. Our repair inserts code into `lookup()` so that it calls an existing initialization function on failure (i.e., before the `return`), re-initializing the static buffer and avoiding

```
#!/bin/sh
# POST data test case for nullhttpd
ulimit -t 5
wget --post-data 'name=claire&submit=submit'
  "http://localhost:$PORT/cgi-bin/hello.pl"
# if the output matches the expected output,
# indicate test passed
diff hello.pl ../hello.pl-expected &&
  echo "passed hello.pl" >> ../list-of-tests
```

```
#!/bin/sh
# Negative test case for nullhttpd
ulimit -t 5
../nullhttpd-exploit -h localhost -p $PORT -t2
wget "http://localhost:$PORT/index.html"
# if the output matches the expected output,
# indicate test passed
diff index.html ../good-index.html-result &&
  echo "passed exploit" >> ../list-of-tests
```

Figure 4.1: Example test cases for **nullhttpd**. The POST test case encodes a positive test. **wget** is a command-line HTTP client; **ulimit** cuts the test off after five seconds. The test assumes that the sandboxed webserver accepts connections on **PORT** and has a copy of **htdocs** that includes **cgi-bin/hello.pl**. Note the oracle comparison using **diff** against **known-good-hello.pl-result** on line 6. In the negative test cases, if the exploit (line 4) crashes the webserver, the request for **index.html** (line 5) will fail.

the segfault. Combined with the explanations of repairs for **nullhttpd** (Section 4.3.1) and **wuftpd** (Section 4.3.5), which include both insertions and deletions, these changes are indicative of repairs involving inserted code.

This experiment demonstrates that GenProg can successfully repair a number of defect types in existing programs, in a reasonable amount of time. These results provide confidence that GenProg is meeting the first of its design goals regarding *expressive power*, and in particular provide support for the claims tested by Hypothesis 1, that without defect- or program-specific information, GenProg can repair at least 5 different defect types and can repair defects in at least least 10 different program types. On this particular benchmark set, it repaired 8 types of defects in 13 types of programs.

4.3 Repair Descriptions

In this section, we begin our investigation of Hypothesis 4, addressing the quality of GenProg-produced patches, by describing several buggy programs and detailing examples of the patches that GenProg generates to repair the bug in question. The purpose of these descriptions is to illustrate the repair process on real-world examples and provide a qualitative sense of the types of repairs that GenProg finds. They also illustrate the test case construction process for several security vulnerabilities. These case studies are also taken from Table 4.1. In each case, we first describe the bug that corresponds to a public vulnerability report; we then describe an indicative patch discovered by GenProg.

4.3.1 nullhttpd: remote heap buffer overflow

The **nullhttpd** webserver is a lightweight multithreaded webserver that handles static content as well as CGI scripts. Version 0.5.0 contains a heap-based buffer overflow vulnerability that allows remote attackers to execute arbitrary code (Chapter 3 adapts this vulnerability for explanatory purposes). **nullhttpd** trusts the **Content-Length** value provided by the user in the HTTP header of POST requests; negative values cause **nullhttpd** to overflow a buffer.

We used six positive test cases that include both GET and POST requests and a publicly available exploit to create the negative test case. The negative test case request crashes the webserver, which is not set to respawn. Figure 4.1 shows parts of one positive test case (on the left) and the single negative test case (on the right). To determine if the attack succeeded, we insert a legitimate request for `index.html` after the exploit; the negative test case fails if the correct `index.html` is not produced.

The actual buffer overflow occurs in the `ReadPOSTData()` function, defined in `http.c`,

```
108 conn[sid].PostData=
109     calloc(conn[sid].dat->in_ContentLength+1024,
110            sizeof(char));
111 pPostData=conn[sid].PostData;
112 ...
113 do {
114   rc=recv(conn[sid].socket,
115           pPostData, 1024, 0); /* overflow! */
116   ...
117   pPostData+=rc;
118 } while ((rc==1024) ||
119           (x<conn[sid].dat->in_ContentLength));
```

The value `in_ContentLength` is supplied by the attacker. However, there is a second location in the program, the `cgi_main()` function on line 267 of `cgi.c`, where POST-data is processed and copied:

```
267 if (conn[sid].dat->in_ContentLength>0) {
268   write(local.out, conn[sid].PostData,
269         conn[sid].dat->in_ContentLength);
270 }
```

The evolved repair changes the high-level `read_header()` function so that it uses the POST-data processing in `cgi_main()` instead of calling `ReadPostData`. The final, minimized repair is 5 lines long. Although the repair is not the one supplied in the next release by human developers—which inserts local bounds-checking code in `ReadPOSTData()`—it both eliminates the vulnerability and retains desired functionality.

This repair also highlights the importance of test suite selection. If set to repair `nullhttpd` *without* a positive test case for POST-data functionality, GenProg generates a repair that disables POST functionality entirely. The POST-processing functionality is associated exclusively with the negative test case, and deleting those statements is the most expedient way to find a variant that passes all such tests. As a quick fix this is not unreasonable, and is safer than the common alarm practice of running in read-only mode [25]. However, including the POST-functionality test case leads GenProg to find the repair described here, that does not remove functionality.

4.3.2 openldap: non-overflow denial of service

The `openldap` server implements the lightweight directory access protocol, allowing clients to authenticate and make queries (e.g., to a company's internal telephone directory). Version 2.3.41 is vulnerable to a denial of service attack. LDAP encodes protocol elements using a lightweight basic encoding rule (BER); non-authenticated remote attackers can crash the server by making improperly formed requests.

The assertion visibly fails in `liblber/io.c`, so we restricted attention to that single file to demonstrate that we can repair program modules in isolation without requiring a whole-program analysis. To evaluate the fitness of a variant `io.c` we copied it in to the `openldap` source tree and ran `make` to rebuild and link the `liblber` library, then applied the test cases to the resulting binary.

The positive test cases consist of an unmodified 25-second prefix of the regression suite that ships with `openldap`. The negative test case was a copy of a positive test case with an exploit request inserted in the middle:

```
perl -e 'print"\xff\xff\xff\x00\x84\x41\x42\x43\x44"' | nc $HOST $PORT
```

The problematic code is around line 522 of `io.c`:

```
516 for (i=1; (char *)p<ber->ber_rwptr; i++) {
517   tag <<= 8;
518   tag |= *p++;
519   if (!(tag & LBER_MORE_TAG_MASK)) break;
520   if (i == sizeof(ber_tag_t)-1) {
521     /* Is the tag too big? */
522     sock_errset(ERANGE); /* !! buggy assert */
523     return LBER_DEFAULT;
524   }
525 }
526 if ((char *)p == ber->ber_rwptr) {
527   /* Did we run out of bytes? */
528   sock_errset(EWOULDBLOCK);
529   return LBER_DEFAULT;
530 }
```

The `for` loop contains both a sanity check and processing for large `ber` tags. The first 127 tag values are represented with a single byte: if the high bit is set, the next byte is used as well, and so on. The repair removes the entire loop (lines 516–524), leaving the "`run out of bytes`" check untouched. This limits the number of BER tags that the repaired `openldap` can handle to 127. A more natural repair would be to fix the sanity check while still supporting multi-byte BER tags. However, only about thirty tags are actually defined for `openldap` requests, so the repair is fine for all `openldap` use cases, and the resulting program passes all the tests.

```
1  POST /hello.php HTTP/1.1
2  Host: localhost:8000
3  Connection: close
4  Content-length: 21213
5  Content-Type: application/x-www-form-urlencoded
6  ...
7  randomly-generated text
8  ...
9  \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
10 SCRIPT_FILENAME/etc/passwd
```

Figure 4.2: Exploit POST request for `lighttpd`. The random text creates a request of the correct size; line 9 uses a fake FastCGI record to mark the end of the data. Line 10 overwrites the execute script so that the vulnerable server responds with the contents of `/etc/passwd`.

4.3.3 lighttpd: remote heap buffer overflow

`lighttpd` is a webserver optimized for high-performance environments; it is used by `YouTube` and `Wikimedia`, among others. In version 1.4.17, the `fastcgi` module, which improves script performance, is vulnerable to a heap buffer overflow that allows remote attackers to overwrite arbitrary CGI variables (and thus control what is executed) on the server machine. In this case, GenProg repaired a dynamically linked shared object, `mod_fastcgi.so`, without touching the main executable.

The positive test cases included requests for static content (i.e., `GET index.html`) and a request to a 50-line CGI Perl script which, among other actions, prints all server and CGI environment variables. The negative test case is the request shown in Figure 4.2, which uses a known exploit to retrieve the contents of `/etc/passwd`—if the file contents are not returned, the test case passes.

The key problem is with the `fcgi_env_add` function, which uses `memcpy` to add data to a buffer without proper bounds checks. `fcgi_env_add` is called many times in a loop by `fcgi_create_env`, controlled by the following bounds calculation:

```
2051 off_t weWant =
2052   req_cq->bytes_in - offset > FCGI_MAX_LENGTH
2053   ? FCGI_MAX_LENGTH : req_cq->bytes_in - offset;
```

The repair modifies this calculation to:

```
2051 off_t weWant =
2052   req_cq->bytes_in - offset > FCGI_MAX_LENGTH
2053   ? FCGI_MAX_LENGTH : weWant;
```

`weWant` is thus uninitialized, causing the loop to exit early on very long data allocations. However, the repaired server can still report all CGI and server environment variables and serve both static and dynamic content.

4.3.4 php: integer overflow

The **php** program is an interpreter for a popular web-application scripting language. Version 5.2.1 is vulnerable to an integer overflow attack that allows attackers to execute arbitrary code by exploiting the way the interpreter calculates and maintains bounds on string objects in single-character string replacements. String processing is common in PHP programs, and the interpreter calculates and maintains bounds on each string object. However, the implementation of the **str_replace** PHP function does not properly compute or bounds-check string lengths, and replacing every character of a large string can cause the length value to overflow.

We manually generated three positive test cases that exercise basic PHP functionality, including iteration, string splitting and concatenation, and popular built-in functions such as **explode**. The negative test case included basic PHP string processing before and after the following exploit code:

```
str_replace("A", str_repeat("B", 65535), str_repeat("A", 65538));
```

A program variant passed this test if it produced the correct output without crashing.

Single-character string replacement replaces every instance of a character (**"A"** in the attack) in a string (65538 **"A"**'s) with a larger string (65535 **"B"**'s). This functionality is implemented by **php_char_to_str_ex**, which is called by function **php_str_replace_in_subject** at line 3478 of file **string.c**:

```
3476  if (Z_STRLEN_P(search) == 1) {
3477    php_char_to_str_ex(
3478     Z_STRVAL_PP(subject), Z_STRLEN_PP(subject),
3479     Z_STRVAL_P(search)[0], Z_STRVAL_P(replace),
3480     Z_STRLEN_P(replace), result,
3481     case_sensitivity, replace_count);
3482  } else if (Z_STRLEN_P(search) > 1) {
3483    Z_STRVAL_P(result) =
3484     php_str_to_str_ex(
3485      Z_STRVAL_PP(subject), Z_STRLEN_PP(subject),
3486      Z_STRVAL_P(search), Z_STRLEN_P(search),
3487      Z_STRVAL_P(replace), Z_STRLEN_P(replace),
3488      &Z_STRLEN_P(result), case_sensitivity,
3489      replace_count);
3490  } else {
3491    *result = **subject;
3492    zval_copy_ctor(result);
3493    INIT_PZVAL(result);
3494  }
```

`php_str_replace_in_subject` uses a macro `Z_STRLEN_P`, defined in a header file, to calculate the new string length. This macro expands to `len + (char_count * (to_len - 1))` on line 3480, wrapping around to a small negative number on the exploitative input. The repair changes lines 3476–3482 to:

```
3476  if (Z_STRLEN_P(search) != 1) {
```

Single-character string replaces are thus disabled, with the output set to an unchanged copy of the input, while multi-character string replaces, performed by `php_str_to_str_ex`, work as before. The `php_str_to_str_ex` function replaces every instance of one substring with another, and is not vulnerable to the same type of integer overflow as `php_char_to_str_ex` because it calculates the resulting length differently. Disabling functionality to suppress a security violation is often a legitimate response in this context: many systems can be operated in a "safe mode" or "read-only mode." Although acceptable in this situation, disabling functionality could have deleterious consequences in other settings; we address this issue in Section 4.4.

4.3.5 wu-ftpd: format string

`wu-ftpd` is an FTP server that allows for anonymous and authenticated file transfers and command execution. Version 2.6.0 is vulnerable to a well-known format string vulnerability. If `SITE EXEC` is enabled, a user can execute a restricted subset of quoted commands on the server. Because the user's command string is passed directly to a `printf`-like function, anonymous remote users gain shell access by using carefully selected conversion characters. Although the exploit is similar in structure to a buffer overrun, the underlying problem is a lack of input validation. GenProg operated on the entire `wu-ftpd` source.

We used five positive test cases (obtaining a directory listing, transferring a text file, transferring a binary file, correctly rejecting an invalid login, and an innocent `SITE EXEC` command). The negative test used a posted exploit to dynamically craft a format string for the target architecture.

The bug is in the `site_exec()` function of `ftpcmd.y`, which manipulates the user-supplied buffer `cmd`:

```
1875  /* sanitize the command-string */
1876  if (sp == 0) {
1877    while ((slash = strchr(cmd, '/')) != 0)
1878      cmd = slash + 1;
1879  } else {
1880    while (sp
1881           && (slash = (char *) strchr(cmd, '/'))
1882           && (slash < sp))
1883      cmd = slash + 1;
1884    for (t = cmd; *t && !isspace(*t); t++) {
1885      if (isupper(*t)) *t = tolower(*t);
```

```
1886    }
1887  }
1888  ...
1889  lreply(200, cmd);
1890  /* !!! vulnerable lreply call */
1891  ...
1892  /* output result of SITE EXEC */
1893  lreply(200, buf);
```

`lreply(x,y,z...)` provides logging output by printing the executing command and providing the return code (`200` denotes success in the FTP protocol). The `lreply(200,cmd)` on line 1889 calls `printf(cmd)` which, with a carefully crafted `cmd` format string, compromises the system. The explicit attempt to sanitize `cmd` by skipping past slashes and converting to lowercase does not prevent format-string attacks. The repair replaces `lreply(200,cmd)` with `lreply(200, (char *)"")`, which disables verbose debugging output on `cmd` itself, but does report the return code and the properly sanitized `site_exec` in `buf` while maintaining required functionality.

These case studies serve as examples of the types of patches that GenProg can find for several real-world security vulnerabilities. We provide two additional examples, and compare them explicitly to human patches for the same errors, in Chapter 5. Although the qualitative investigations in this section is encouraging, it does not systematically address the important issue of repair quality; we turn to this question in the next section.

4.4 Repair Quality

We now approach Hypothesis 4 quantitatively. Industrial practitioners and researchers alike take several approaches to evaluate the quality of a candidate source code change, though no one metric or measure is currently suitable to fully address the concern. We thus begin this section by outlining a framework for quantitatively evaluating repair quality (Section 4.4.1). This framework motivates a repair quality experimental setup (Section 4.4.2 and Section 4.4.3). The remainder of the section presents experimental results that evaluate several key aspects of repair quality; we road map those experiments in Section 4.4.3.

4.4.1 Repair quality experimental framework

This section identifies the core ways a repair may affect the quality of a system. This discussion motivates our framework for experimentally evaluating the quality of several GenProg repairs (including benchmark selection).

A source code change may affect either a program's functional requirements, its non-functional requirements (such as efficiency or runtime), or both.

Functional requirements. GenProg's reliance on positive test cases provides an important check against patches that reduce service relative to a program's functional requirements. However, if the input test suites are inadequate, GenProg's output may compromise a program's service. The input test suites may be inadequate either because the positive test cases are inadequate—failing to sufficiently test or exercise required program functionality—or because the negative test cases are inadequate—failing to fully or properly encode the defect under repair. Inadequate positive tests may lead to:

(P1) **Reduction of program functionality.** A repair might reduce functionality by deleting behavior that is buggy only in some instances (those tested by the negative test cases), but under-tested by the positive test cases. Section 4.3.1 provided an example from `nullhttpd`: without a positive test case for POST functionality, GenProg finds a repair that removes that functionality entirely.

(P2) **Introduction of new vulnerabilities.** One way that human fixes can be faulty is by introducing new failure modes or attack vectors; these types of vulnerabilities are difficult to detect with test suites that encode functional requirements (which is one reason there is considerable research dedicated to finding them automatically).

The risks posed by inadequate negative tests include:

(N1) **Fragile repairs.** A repair might be *fragile*, in that it applies exclusively to the faulty input used in a negative test case without addressing the underlying cause of the defect. For example, the negative test case for `nullhttpd` exploits the underlying vulnerability by sending a POST request with a Content-Length of -800. A patch that caused `nullhttpd` to reject POST requests with Content-Lengths of -800 would be addressing the *symptom* of the failure encoded in the test case, but not the underlying cause, which is improper handling of negative request lengths. We prefer a patch to `nullhttpd` that causes the webserver to safely reject all such requests, not just those that correspond exactly to the one specified by the negative test case; the GenProg patch described in Section 4.3.1 is satisfactory by this definition.

(N2) **Repairs of non-defects.** A badly-constructed negative test case may fail to identify truly buggy behavior. This is an especial risk if GenProg is integrated with automatic bug-finding tools, but imprecise bug reports with inaccurate steps to reproduce may also lead to a poorly-written test case. A poorly-written or inaccurate negative test case may lead to undesirable search results, reducing or arbitrarily modifying existing program functionality.

Non-functional requirements. Beyond the risks posed by inadequate test suites, the execution-based testing upon which GenProg relies may not adequately focus on non-functional requirements, including system runtime, efficiency, or throughput. As a result, a patch generated in response to such tests may unacceptably compromise program performance. Additionally, the use of test cases exclusively to define acceptability admits the possibility of repairs that degrade the

quality of the design of a system or make a system more difficult to maintain, concerns that have been evaluated in related work [50].

The experiments in this section in general proceed by subjecting several of the programs in Table 4.1 to additional test suites (we provide specifics in the next subsection). The test suites are targeted to evaluating one of the dangers posed by low-quality repairs, and the metrics used to evaluate the results of running them address both functional and non-functional performance. The test suites are:

(T1) **Held-out indicative workloads** to evaluate the impact of the patches on existing functionality. Indicative workloads correspond to the real-world usage of a system, and thus its actual expected functionality. Behavior on these workloads pre- and post- patch can be evaluated both in terms of *correctness* (expected output) and *performance*. A patched program should exhibit the same performance on an indicative workload that the unpatched program does: it should compute the same correct results in the same amount of time (while avoiding the bug). These measures evaluate both the risks of reduced functionality as well as the risks of degraded performance relative to certain non-functional requirements, like runtime or efficiency.

When used to evaluate patches produced using adequate negative test cases, these workloads allow us to evaluate the risks posed by inadequate positive test suites, concern P1 above. When used to evaluated patches produced in response to low-quality negative test cases, these workloads help address concern N2.

(T2) **Black-box fuzz testing inputs** to gain confidence that a patch does not introduce new vulnerabilities. This corresponds to one approach industrial practitioners use to vet security-critical patches: testing patched software against a large number of randomly-generated inputs. A patched program should fail no new fuzz inputs as compared to the unpatched program. This test suite helps address concern P2.

(T3) **Exploit variants generated by fuzzing techniques** to evaluate whether a patch addresses the underlying cause of a vulnerability as opposed to the symptom encoded in the negative test case. Fuzz testing research has developed techniques to generate new exploitative inputs based on an existing exploitative input, designed to be distinct from the original input while taking advantage of the same attack vector. A patched program should reject all exploitative variants that the unpatched program passes. This type of test suite helps address concern N1.

These concerns and the associated metrics motivate several design requirements in a framework that quantitatively evaluates repair quality. First, the benchmarks should admit large, indicative workloads as well as testing via existing fuzz testing tools. The benchmarks should also admit performance evaluation relative to non-functional requirements, like efficiency. Second, in a real-time deployment, GenProg can impact performance both via the overhead imposed by a low-quality patch and via the overhead of running GenProg *itself*; our experimental framework should allow us to

measure both. Finally, our experimental framework should allow us to measure the potential effects of badly-constructed negative test cases.

The current claim of *automated* program repair relies on manual initialization and dispatch of GenProg. In principle, however, automated detection techniques could signal the repair process to complete the automation loop. To address the concerns above, we integrate GenProg with an automated intrusion detection system (IDS) to form a closed-loop system that monitors a running server, detects errors or anomalies, and automatically signals GenProg to generate needed repairs. Such a system enables exactly the types of quantitative repair experiments we require. First, existing IDS techniques are particularly well-developed for the monitoring of webservers. This is beneficial experimentally first because it identifies three case study patches from Table 4.1 that are associated with or can be tested in the context of webservers: `lighttpd`, `php`, and `nullhttpd`. Second, webservers admit the types of workloads and overhead metrics these experiments require, and are readily served by existing off-the-shelf fuzz testing systems. Third, as discussed in Chapter 2, IDS techniques can produce false positives; that is, they can report that a request is dangerous when it is in fact benign. Such false positives allow us to evaluate the risk posed by poorly-designed negative test cases.

Thus, given suitable workloads and fuzz-testing tools, we use a prototype closed-loop system as a grounded framework to explore repair quality across the several dimensions identified in Section 4.4.1. We next outline this system (Section 4.4.2) and our concrete experimental setup (Section 4.4.3).

4.4.2 Closed-Loop System Overview

This subsection describes a prototype, proof-of-concept closed-loop repair system, motivated in Section 4.4.1. We use this system to quantify the quality of several repairs generated in the experiments in this chapter; we evaluate them against the concerns and using the metrics presented and motivated in the previous subsection.

Our prototype closed-loop repair system has two requirements beyond the input required by GenProg: (1) anomaly detection in near-real time, to provide a signal to launch the repair process and (2) the ability to record and replay system input [208] so we can automatically construct a negative test case. Anomaly detection could be provided by existing behavior-based techniques that run concurrently with the program of interest, operating at almost any level (e.g., by monitoring program behavior, examining network traffic, using saved state from regular checkpoints, etc.). Our prototype adopts an intrusion-detection system (IDS) that detects suspicious HTTP requests based on request features [209]. In a pre-processing phase, the IDS learns a probabilistic finite state machine model of normal requests using a large training set of legitimate traffic. After training, the model labels subsequent HTTP requests with a probability corresponding to "suspiciousness." We combine this IDS with lightweight external checks on implicit program requirements, such as whether or not it has segfaulted.

Given these components, the system works as follows. While the webserver is run normally and exposed to untrusted inputs from the outside world, the IDS checks for anomalous behavior, and the system stores program state and each input while it is being processed. When the IDS detects an anomaly, the program is suspended, and GenProg is invoked to repair the suspicious behavior. The negative test case is constructed from the IDS-flagged input: a variant is run in a sandbox on the input with the program state stored from just before the input was detected. If the variant terminates successfully without triggering the IDS or the external checks on program behavior, the negative test case passes; otherwise, it fails. The positive tests consist of standard system regression tests. For the purpose of these experiments, we use the tests described in Section 4.1 to guide the repair search, and add new, large indicative workloads to evaluate the effect of the repair search and deployment on several benchmarks.

If a patch is generated, it can be deployed immediately. If GenProg cannot locate a viable repair within the time limit, subsequent identical requests should be dropped and an operator alerted. While GenProg runs, the system can either refuse requests, respond to them in a "safe mode" [25], or use any other technique (e.g., fast signature generation [210]) to filter suspicious requests. Certain application domains (e.g., supply chain management requests, banking or e-commerce) support buffering of requests received during the repair procedure, so they can be processed later.

The effects of the proposed system on a running program depend on the anomaly detector's misclassification rates (false positives/negatives) and the efficacy of the repair method. A patch that degrades functionality not guaranteed by the positive tests, produced in response to a correctly detected vulnerability, can lead to the loss of legitimate requests or, in the worst case, new vulnerabilities. A false positive identified by the IDS is an example of a poorly-motivated or -constructed negative test case; patches produced in response to such false positives may also impede functionality.

The remainder of the section evaluates concrete repair quality concerns within the closed-loop framework. We provide a road map for these experiments in the next subsection, which presents benchmarks, workloads, and success metrics for those experiments.

4.4.3 Repair quality experimental setup

The high-level goal of these experiments is to quantitatively evaluate the quality of automatically-generated repairs. As motivated in Section 4.4.1, we evaluate the effect of the repairs in terms of their impact on both functional and non-functional requirements, and we do so in the context of a closed loop system for webserver vulnerability detection and repair (as motivated in Section 4.4.1 and presented in Section 4.4.2).

We focus on three of our benchmarks that consist of security vulnerabilities in long-running servers: `lighttpd`, `nullhttpd`, and `php`. To construct indicative workloads (T1), we obtained workloads and content layouts from the University of Virginia CS department webserver. To evaluate repairs to the `nullhttpd` and `lighttpd` webservers, we

used a workload of 138,226 HTTP requests spanning 12,743 distinct client IP addresses over a 14-hour period on November 11, 2008. To evaluate repairs to `php`, we obtained the room and resource reservation system used by the University of Virginia CS department, which features authentication, graphical animated date and time selection, and a `mysql` back-end. It totals 16,417 lines of PHP, including 28 uses of `str_replace` (the subject of the `php` repair), and is a fairly indicative three-tier web application. We also obtained 12,375 requests to this system, spanning all of November 11, 2008. Recall that the `php` repair deletes functionality; we use this workload to evaluate the effect of such a repair (concern P1).

In all cases, we label a request "successful" if the correct (bit-for-bit) data was returned to the client before that client started a new request. Success requires both correct *output* (a functional requirement) and correct *response time* (a non-functional requirement). To evaluate the overhead of a low-quality repair, we measure and compare the number of successful requests each program processed before and after a repair. To evaluate repair time overhead, we assume a worst-case scenario in which the same machine is used both for serving requests and repairing the program, and in which all incoming requests are dropped (i.e., not buffered) during the repair process. We then measure the number of requests the program drops during the repair process.

Our test machine contains 2 GB of RAM and a 2.4 GHz dual-core CPU. To perform a more controlled experiment and avoid masking repair cost, we uniformly sped up the workloads until the server machine was at 100% utilization (and additional speedups resulted in dropped packets). Similarly, to remove network latency and bandwidth considerations, we ran servers and clients on the same machine. As a result of these two controls, any extra computational burden imposed by GenProg or its patches should result in dropped or delayed requests. This gives greater confidence that the results of this experiment will generalize even to deployments that do not have unused server or network capacity.

We use an off-the-shelf fuzz testing tool to generate random inputs to evaluate concerns P2 and N1 (T2 and T3), as described in Section 4.4.1. Comparing behavior pre- and post-repair can suggest whether a repair has introduced new errors, and whether the repair generalizes.

The remainder of this section evaluates GenProg and the repairs it generates for these webserver-based programs in the context of the proof-of-concept closed-loop system for webserver-based programs, with several experimental questions. These questions, and their associated subsection and quality concerns, are:

1. What is the performance impact of repair time on a real, running system? (Section 4.4.4, P1)

2. What is the performance impact of repair quality, especially with respect to a functionality-reducing repair? (Section 4.4.5, P1)

3. Do the repairs appear to introduce new vulnerabilities, as measured by large-scale black-box fuzz testing? (Section 4.4.6, P2)

Program	Repair Made?	Requests Lost to Repair Time	Requests Lost to Repair Quality	Fuzz Test Failures: Generic Before	Generic After	Exploit Before	Exploit After
`nullhttpd`	Yes	$2.38\% \pm 0.83\%$	$0.00\% \pm 0.25\%$	0	0	10	0
`lighttpd`	Yes	$2.03\% \pm 0.37\%$	$0.03\% \pm 1.53\%$	1410	1410	9	0
`php`	Yes	$0.12\% \pm 0.00\%$	$0.02\% \pm 0.02\%$	3	3	5	0
Quasi False Pos. 1	Yes	$7.83\% \pm 0.49\%$	$0.00\% \pm 2.22\%$	0	0	—	
Quasi False Pos. 2	Yes	$3.04\% \pm 0.29\%$	$0.57\% \pm 3.91\%$	0	0	—	
Quasi False Pos. 3	No	$6.92\% \pm 0.09\%$	—	—			

Table 4.3: Closed-loop repair system evaluation. Each row represents a different repair scenario and is separately normalized so that the pre-repair daily throughput is 100%. The `nullhttpd` and `lighttpd` rows show results for true repairs. The `php` row shows the results for a repair that degrades functionality. The False Pos. rows show the effects of repairing three intrusion detection system false positives on `nullhttpd`. The number after $\pm$ indicates one standard deviation. "Lost to Repair Time" indicates the fraction of the daily workload lost while the server was offline generating the repair. "Lost to Repair Quality" indicates the fraction of the daily workload lost after the repair was deployed. "Generic Fuzz Test Failures" counts the number of held-out fuzz inputs failed before and after the repair. "Exploit Failures" measures the held-out fuzz exploit tests failed before and after the repair.

4. Do the repairs address the underlying vulnerability, as measured by variant exploit fuzz testing? (Section 4.4.6, N1)

5. What are the costs associated with badly-constructed negative test cases, as measured by the costs of intrusion-detection system false positives? (Section 4.4.7, N2)

All five of these experimental questions serve to substantiate our claim that GenProg patches are of sufficient quality for real-world deployment. Hypothesis 4 summarized these metrics and their success criteria; it is the overarching hypothesis for these experiments.

4.4.4 The Cost of Repair Time

This subsection evaluates our first experimental question by measuring the overhead of the repair process itself. We measure the number of requests from the indicative workloads the unmodified programs successfully handle. Next, we generated the repair, noting the requests lost during the time taken to repair on the server machine. Figure 4.3 summarizes the results. The "Requests Lost To Repair Time" column shows the requests dropped during the repair as a fraction of the total number of successful requests served by the original program. To avoid skewing relative performance by the size of the workload, the numbers have been normalized to represent a single day containing a single attack. Note that the absolute speed of the server is not relevant here: a server machine that was twice as fast overall would generate the repair in half the time, but would also process requests twice as quickly. Fewer than 8% of daily requests were lost while the system was offline for repairs. Buffering requests, repairing on a separate machine, or using techniques such as signature generation could reduce this overhead.

4.4.5 The Cost of Repair Quality

This subsection evaluates our second experimental question by measuring the patches' impact on performance and throughput. The "Requests Lost to Repair Quality" column of Figure 4.3 quantifies the effect of the generated repairs on program throughput. This row shows the difference in the number of requests that each benchmark could handle before and after the repair, expressed as a percentage of total daily throughput. The repairs for **nullhttpd** and **lighttpd** do not noticeably affect their performance. Recall that the **php** repair degrades functionality by disabling portions of the **str_replace** function. The **php** row of Figure 4.3 shows that this low quality (loss of functionality) repair does not strongly affect system performance. Given the low-quality repair's potential for harm, the low "Lost" percentage for **php** is worth examining. Of the reservation application's 28 uses of **str_replace**, 11 involve replacements of multi-character substrings, such as replacing `'--'` with `'- -'` for strings placed in HTML comments. Our repair leaves multi-character substring behavior unchanged. Many of the other uses of **str_replace** occur on rare paths. For example, in:

```
$result = mysql_query($query) or
  die("Query failed : " . mysql_error());
while($data=mysql_fetch_array($result,ASSOC))
  if (!$element_label)
    $label = str_replace('_',' ',$data['Fld']);
  else
    $label = $element_label;
```

str_replace is used to make a form label, but is only invoked if another variable, **element_label**, is **null**. Other uses replace, for example, underscores with spaces in a form label field. Since the repair causes single-character **str_replace** to perform no replacements, if there are no underscores in the field, then the result remains correct. Finally, a few of the remaining uses were for SQL sanitization, such as replacing `',` with `"','"`. However, the application also uses **mysql_real_escape_string**, so it remains safe from such attacks.

4.4.6 Repair Generality and Fuzzing

The experiments in the previous subsections suggest that GenProg repairs do not impair legitimate requests, an important component of repair quality. This section evaluates experimental questions 3 and 4 using off-the-shelf fuzz testing.

First, repairs must not introduce new flaws or vulnerabilities, even when such behavior is not tested by the input test cases. To this end, Microsoft requires that security-critical changes be subject to 100,000 fuzz inputs [211] (i.e., randomly generated structured input strings). Similarly, we used the **SPIKE** black-box fuzzer from http://immunitysec.com to generate 100,000 held-out fuzz requests using its built-in handling of the HTTP protocol.

The "Generic" column in Figure 4.3 shows the results of supplying these requests to each program. Each program failed no additional tests post-repair: for example, `lighttpd` failed the same 1410 fuzz tests before and after the repair.

Second, a repair must do more than merely memorize and reject the exact attack input: it must address the underlying vulnerability. To evaluate whether the repairs generalize, we used the fuzzer to generate 10 held-out variants of each exploit input. The "Exploit" column shows the results. For example, `lighttpd` was vulnerable to nine of the variant exploits (plus the original exploit attack), while the repaired version defeated all of them (including the original). In no case did GenProg's repairs introduce any errors that were detected by 100,000 fuzz tests, and in every case GenProg's repairs defeated variant attacks based on the same exploit, showing that the repairs were not simply fragile memorizations of the input.

The issue of repair generality extends beyond the security examples shown here. Note that because this particular experiment only dealt with the repair of security defects, fuzz testing was more applicable than it would be in the general case. Establishing that a repair to a generic software engineering error did not introduce new failures or otherwise "overfit" could also be accomplished with held out test cases or cross-validation.

4.4.7 Cost of Intrusion Detection False Positives

Finally, we examine experimental question 5, which concerns the dangers of low-quality negative test cases. We evaluate this concern by examining the effect of IDS false positives when used as a signal to GenProg. We trained the IDS on 534109 requests from an independent dataset [209]; this process took 528 seconds on a machine with quad-core 2.8 GHz and 8 GB of RAM. The resulting system assigns a score to each incoming request ranging from 0.0 (anomalous) to 1.0 (normal). However, the IDS perfectly discriminated between benign and exploitative requests in the testing workloads (no false negatives or false positives), with the threshold of 0.02 suggested in the related research. Therefore, to perform these experiments, we randomly selected three of the lowest-scoring normal requests (closest to being incorrectly labeled anomalous) and attempted to "repair" `nullhttpd` against them, using the associated requests as input and a `diff` against the baseline result for the negative test case; we call these requests quasi-false positives (QFPs). The "False Pos." rows of Figure 4.3 show the effect of time to repair and requests lost to repair when repairing these QFPs.

QFP #1 is a malformed HTTP request that includes quoted data before the GET:

```
"[11/Nov/2008:12:00:53 -0500]" GET /people/modify/update.php HTTP/1.1
```

The GenProg repair changed the error response behavior so that the response header confusingly includes `HTTP/1.0 200 OK` while the user-visible body retains the correct `501 Not Implemented` message, but with the color-coding stripped. The header inclusion is ignored by most clients; the second change affects the user-visible error

message. Neither causes the webserver to drop additional legitimate requests, and Figure 4.3 shows no significant loss due to repair quality.

QFP #2 is a `HEAD` request; such requests are rarer than `GET` requests and only return header information such as last modification time. They are used by clients to determine if a cached local copy suffices:

```
HEAD /~user/mission_power_trace_movie.avi HTTP/1.0
```

The repair changes the processing of `HEAD` requests so that the `Cache-Control: no-store` line is omitted from the response. The `no-store` directive instructs the browser to store a response only as long as it is necessary to display it. The repair thus allows clients to cache pages longer than might be desired. It is worth noting that the `Expires: <date>` also included in the response header remains unchanged and correctly set to the same value as the `Date: <date>` header (also indicating that the page should not be cached), so a conforming browser is unlikely to behave differently. Figure 4.3 indicates negligible loss from repair quality.

QFP #3 is a relatively standard HTTP request:

```
GET /~lcc-win32/lccwin32.css HTTP/1.1
```

GenProg fails to generate a repair within one run (240 seconds) because it cannot generate a variant that is successful at `GET index.html` (one of the positive test cases) but fails the almost identical QFP #3 request. Since no repair is deployed, there is no subsequent loss to repair quality.

These results quantitatively test by our hypothesis regarding the quality of GenProg repairs by evaluating them on held-out workloads, black box fuzz inputs, and exploitative variants. These experiments support the claim that GenProg produces repairs that address the given errors and do not compromise other functionality. It appears that the time taken to generate these repairs is reasonable and does not unduly influence real-world program performance. Finally, the experiments suggest that the danger from low-quality negative test cases is lower than that of low-quality repairs from inadequate test suites, but that both limitations are manageable. These results suggest that GenProg's applicability may generalize to deployments that have imperfect IDS or bug detection or imperfect human operators, because it is robust in the face of inadequate test suites of several varieties.

4.5 Discussion and Summary

This chapter provides evidence to substantiate our claims about GenProg's expressive power specifically as it relates to our first hypothesis, that is, the different types of bugs and the different types of programs that GenProg can repair without *a priori* knowledge (Section 4.2). Those benchmarks and associated repair results admit qualitative descriptions of the types of repairs that GenProg has found (Section 4.3), and motivate a discussion about the factors that influence automatic repair quality (Section 4.4.1). We developed a novel framework for quantitatively evaluating the quality of GenProg repairs in the context of a closed-loop system for detecting and repairing security vulnerabilities (Section 4.4.2).

This allowed us to evaluate our second hypothesis about GenProg's utility, and provided evidence that the patches it generates are of acceptable quality (Section 4.4.3, and those subsections that followed). These experiments indicated that GenProg and its patches have negligible negative impact on non-functional performance requirements (specifically efficiency); the costs associated with inadequate test suites (either positive or negative) are low; the repairs generalize without introducing new vulnerabilities; and the approach may be viable when applied to real programs with real workloads, even in the face of integration with imperfect tools or in the hands of imperfect operators.

However, there are several limitations with respect to these results that are worth noting:

Non-determinism. GenProg relies on test cases to encode both an error to repair and important functionality. Some properties are difficult or impossible to encode using test cases, such as nondeterministic properties (e.g., race conditions). We note, however, that multithreaded programs, such as `nullhttpd`, can already be repaired, so long as the bug itself is deterministic. This limitation could be mitigated by running each test case multiple times, incorporating scheduler constraints into the GP representation and allowing a repair to contain both code changes and scheduling directives, or making multithreaded errors deterministic [212]. There are certain other classes of properties, such as liveness, fairness and non-interference, that cannot be disproved with a finite number of execution examples; it is not clear how to test or patch non-interference information flow properties using our system.

Experimental validity. There exist several threats to the validity of our results. We focus in particular on the selection of GenProg parameters and the generality of the benchmarks used. For the former concern, many of the parameters in these experiments (e.g. Section 4.1) were heuristically chosen based on empirical performance. We return to this issue in Chapter 6 and more thoroughly investigate the effects of representations, operators and parameters on GenProg's performance.

Second, these benchmarks, and thus the patterns seen in Figure 4.2 and Figure 4.3, might not generalize to other types of defects or other programs, representing a threat to the external validity of the results. The experiments focus particularly on security-critical vulnerabilities in open-source software, which may not be indicative of all programs or errors found in industry. To mitigate this threat, we attempted to select a variety of benchmarks and errors on which to evaluate GenProg, and restrict our claims in this chapter to those regarding expressive power with respect to the variety of bug types that GenProg has been shown to successfully repair. We address broader external validity with respect to the types of bugs that humans address in practice in the next chapter.

Chapter 5

GenProg is human competitive

CLOUD computing, in which virtualized processing power is purchased cheaply and on-demand, is becoming commonplace [90]. Such resources are increasingly used to alleviate software engineering costs, especially those related to testing and development [89]. In this chapter, we leverage these publicly-available resources to enable a large-scale evaluation of GenProg's scalability and expressive power. Specifically, we examine the dual research questions: "What fraction of real-world bugs can GenProg repair?" and "How much does it cost to repair a bug with GenProg?" Answering these questions requires experimental and performance innovations, specifically to develop an indicative, generalizable set of benchmark bugs in large programs, and to evaluate GenProg's costs in human-comparable time and monetary terms.

The previous chapter provided evidence to substantiate our claims about GenProg's expressive power specifically as it relates to our first hypothesis, that is, the number of different types of bugs and programs that GenProg can repair without *a priori* knowledge. We used those benchmarks and associated repair results to frame a discussion and evaluation of the quality of GenProg repairs. The results in the previous chapter, in other words, serve as an existence proof of the viability automatic patch generation using stochastic search.

This chapter completes our argument by evaluating GenProg explicitly as compared to the human developers it is intended to assist. Our definition of *expressive power* is two-fold: we call an automatic program repair technique *expressive* if it can apply generally to many different types of bugs in many different types of programs, and also if it can *repair many of the bugs that human developers can and do fix in industrial practice*. This includes bugs in large programs with many test cases; GenProg must be scalable to fulfill its design goals. Specifically, this chapter evaluates the following hypotheses, presented first in Chapter 1:

Hypothesis 2: GenProg can repair at least 50% of defects that humans developers fix in practice.

and

Hypothesis 3: GenProg can repair bugs in programs of up to several million lines of code, and associated with up to several thousand test cases, at a time and economic cost that is human competitive.

To test these hypotheses, we establish a framework for explicitly comparing an automated repair technique with the humans it is intended to assist. This comparison is designed to substantiate our claim that GenProg is human competitive. This includes a definition of experimental metrics as well as the generation of a large dataset of bugs in programs that are indicative of those that human developers address in practice. We then use GenProg to repair as many of those bugs as possible, in a way that enables grounded cost measurements, and report the results.

This chapter begins with a motivation for the metrics we use to evaluate human-comparable costs and benefits (Section 5.1). The remainder of the chapter is structured to follow its contributions, which are:

- A novel experimental framework that enables the grounded evaluation of the human competitive cost of a technique for automatic bug repair (Section 5.2). Our key performance insight is to use off-the-shelf cloud computing as a framework for exploiting search-space parallelism as well as a source of grounded cost measurements.

- A novel systematic procedure for generating an indicative set of real-world defects from open-source programs, that includes all reproducible bugs in a time window and where the defects considered were important enough for developers to test and fix manually (Section 5.3). Our key experimental insight is to search version control histories exhaustively, focusing on open-source C programs, to identify revisions that correspond to human bug fixes as defined by the program's most current test suite.

- The ManyBugs benchmark suite, a set of 105 defects from 5.1 million lines of code from open-source projects equipped with 10,193 test cases. This suite enables an evaluation that provides at least two orders of magnitude size increase over those found in previous work on at least one of three dimensions: code size (as compared to AutoFix-E [171]), test cases (as compared to ClearView [13]), and defects (as compared to AFix [181]), in addition to being strictly larger than each of those previous projects on each of the those metrics separately.

- Results showing that GenProg repairs 55 of those 105 defects, directly testing the second component of expressive power in terms of percentage of real-world defects repaired, and demonstrating scalability to these real-world systems and their indicative test suites (Section 5.4). We also analyze two additional automatically-generated patch examples to extend our illustration of GenProg's repair approach, similar to those patches qualitatively explained in the previous chapter (Section 5.6).

- An analysis of the costs of those repairs that evaluates them as compared to human developer repair costs (Section 5.4.2). Because our experiments were conducted using cloud computing and virtualization, any organization could pay the same rates we did and reproduce our results for $403 — or $7.32 per successful run

on this dataset. A successful repair takes 96 minutes of wall-clock time, on average, while an unsuccessful run takes 11.2 hours, including cloud instance start up times.

These results are comparable and competitive with human bug fixing costs (Section 5.4.2). We also evaluate GenProg's utility and costs as part of two alternative use cases, further substantiating our claims of GenProg's utility (Section 5.5).

5.1 Motivation

This section motivates monetary cost, success rate and wall-clock turnaround time as important evaluation metrics for an automatic bug repair technique that seeks to be competitive with human developers.

The rate at which software bugs are reported has kept pace with the rapid rate of modern software development, as discussed in Chapter 1 and Chapter 2. In light of this problem, many companies have begun offering *bug bounties* to outside developers, paying for candidate repairs to bugs in their open-source code. Well-known companies such as Mozilla[1] and Google[2] offer significant rewards for security fixes, with bounties raising to thousands of dollars in "bidding wars."[3] We briefly describe such bug bounties to highlight that paying strangers to submit untrusted patches is part of the current state of the art, and thus provide context and metrics for naturally comparing GenProg to humans.

Although security bugs command the highest prices, more wide-ranging bounties are available. Consider the more indicative case of Tarsnap.com,[4] an online backup provider. Over a four-month period, Tarsnap paid $1,625 for fixes for issues ranging from cosmetic errors (e.g., typos in source code comments), to general software engineering mistakes (e.g., data corruption), to security vulnerabilities. Of the approximately 200 candidate patches submitted to claim various bounties, about 125 addressed spelling mistakes or style concerns, while about 75 addressed more serious issues, classified as "harmless" (63) or "minor" (11). One issue was classified as "major." Developers at Tarsnap confirmed corrections by manually evaluating all submitted patches. If we treat the 75 non-trivial repairs as true positives (38%) and the 125 trivial reports as overhead, Tarsnap paid an average of $21 for each non-trivial repair and received one about every 40 hours. Despite the facts that the bounty pays a small amount even for reports that do not result in a usable patch and that about 84% of all non-trivial submissions fixed "harmless" bugs, the final analysis was: "Worth the money? Every penny."[5]

Bug bounties suggest that the need for repairs is so pressing that companies are willing to pay for outsourced candidate patches even though repairs must be manually reviewed, most are rejected, and most accepted repairs are for

[1] http://www.mozilla.org/security/bug-bounty.html $3,000/bug
[2] http://blog.chromium.org/2010/01/encouraging-more-chromium-security.html $500/bug
[3] http://www.computerworld.com/s/article/9179538/Google_calls_raises_Mozilla_s_bug_bounty_for_Chrome_flaws
[4] http://www.tarsnap.com/bugbounty.html
[5] http://www.daemonology.net/blog/2011-08-26-1265-dollars-of-tarsnap-bugs.html

low-priority bugs. These examples suggest that relevant success metrics for a repair scheme include (1) the fraction of queries that produce code patches, (2) monetary cost for each patch, and (3) wall-clock time cost. In this chapter, we evaluate GenProg so as to allow direct measurement of these costs, with a hypothetical use case similar to that of the outsourced "bug bounty hunters."

5.2 Experimental Design

While the benchmarks in the previous chapter are selected with an eye to variety, in this chapter, we seek a set of benchmarks that are indicative of those that human developers repair "in the wild." In other words, we want a set that can substantiate GenProg's expressive power in terms of *fraction of real-world bugs* it can repair. In particular, Section 5.1 motivated three metrics for human competitive patch generation: (1) the fraction of queries that produce code patches, (2) monetary cost for each patch, and (3) wall-clock time cost.

We simultaneously evaluate scalability and metric (1) by systematically generating a large, indicative set of benchmark defects in large open-source programs (the ManyBugs benchmark set, described in the next section), and then running GenProg on each defect in the set to observe how many it successfully repairs in a controlled environment.

Metrics (2) and (3) require additional experimental innovation. Because we target a human competitive application, we require a framework in which to measure time and monetary cost that is indicative of those incurred in industrial practice. We therefore ran these experiments using publicly-available cloud computing resources, which enable the purchase of virtual machine compute time at hourly rates. Cloud computation is increasingly employed in industrial software engineering practice, especially for testing [89]; we propose to extend it to automatic repair, and use it as a grounded platform framework that allows us to measure the time- and monetary- cost of our repair experiments in a publicly-comparable way.

We therefore ran 10 GenProg *trials* in parallel for each bug. Each individual was mutated exactly once each generation, crossover is performed once on each set of parents, and 50% of the population is retained (with mutation) on each generation (known as elitism). We chose $PopSize = 40$ and a maximum of 10 generations for consistency with our previous experiments. Each trial was terminated after 10 generations, 12 hours, or when another search found a repair, whichever came first. The twelve hour time limit is motivated by a hypothetical real-world use case: a developer might run GenProg overnight to attempt to address the defects that remain at the end of the workday, with the intention of inspecting candidate patches the next morning [213]. SampleFit returns 10% of the test suite for all benchmarks. We also ran each trial up to 10 generations or 12 hours, to record additional statistics (e.g., how many distinct patches can be produced); we analyze these results in Section 5.5

Program	LOC	Tests	Defects	Description
`fbc`	97,000	773	3	legacy coding
`gmp`	145,000	146	2	precision math
`gzip`	491,000	12	5	data compression
`libtiff`	77,000	78	24	image processing
`lighttpd`	62,000	295	9	web server
`php`	1,046,000	8,471	44	web programming
`python`	407,000	355	11	general coding
`wireshark`	2,814,000	63	7	packet analyzer
total	**5,139,000**	**10,193**	**105**	

Table 5.1: Subject C programs, test suites and historical defects: Tests were taken from the most recent version available in May, 2011; Defects are defined as test case failures fixed by developers in previous versions.

We ran these repair experiments in Amazon's EC2 cloud computing infrastructure. Each trial was given a "high-cpu medium (c1.medium) instance" with two cores and 1.7 GB of memory.[6] Simplifying a few details, at the date the experiment was conducted, the virtualization could be purchased as *spot instances* at $0.074 per hour but with a one hour start time lag, or as *on-demand instances* at $0.184 per hour. These August–September 2011 prices summarize CPU, storage and I/O charges; prices since then have decreased steadily.[7]

5.3 The ManyBugs Benchmark Suite

This section describes the ManyBugs benchmark suite and our procedure for generating it. Table 5.1 summarizes the programs used in our experiments. We selected these benchmarks by first defining predicates for acceptability, and then examining various program repositories to identify first, acceptable candidate programs that passed the predicates; and second, all reproducible bugs within those programs identified by searching backwards from the checkout date (late May, 2011).

At a high level, we seek benchmarks comprised of an unbiased set of programs and defects that can run in our experimental framework and is indicative of "real-world usage." We require that *subject programs* contain sufficient C source code, a version control system, a test suite of reasonable size, and a set of suitable subject defects. We only used programs that could run without modification under cloud computing virtualization, which limited us to programs amenable to such environments. We require that *subject defects* be reproducible and important.

To produce this set, we searched systematically through each program's source history, looking for revisions that caused the program to pass test cases that it failed in a previous revision. Such a scenario corresponds to a human-written repair for the bug corresponding to the failing test case. This approach succeeds even in projects without explicit

[6] http://aws.amazon.com/ec2/instance-types/
[7] http://aws.amazon.com/ec2/pricing/

bug-test links (cf. [214]), and it ensures that benchmark bugs are important enough to merit a human fix and to affect the program's test suite.

Put formally: a candidate subject program is a software project containing at least 50,000 lines of C code, 10 viable test cases, and 300 versions in a revision control system. We consider all *viable versions* of a program, defined as a version that checks out and builds unmodified on 32-bit Fedora 13 Linux (a lowest common denominator OS available on the Amazon EC2 cloud computing framework). A program *builds* if it produces its primary executable, regardless of the exit status of `make`.

We define *test cases* to be the smallest atomic testing units for which individual pass or fail information is available. For example, if a program has 10 "major areas" which each contain 5 "minor tests" and each "minor test" can pass or fail, we say that it has 50 test cases. We define a *viable test case* as a test that is reproducible, non-interactive, and deterministic in the cloud environment (over at least 100 trials). We write $testsuite(i)$ to denote the set of viable test cases passed by viable version i of a program. We use all available viable tests, even those added *after* the version under consideration. We exclude programs with test suites that take longer than one hour to complete in the cloud environment.

We say that a *testable bug* exists between viable versions i and j of a subject program when:

1. $testsuite(i) \subsetneq testsuite(j)$ and
2. there is no $i' > i$ or $j' < j$ with the $testsuite(j) - testsuite(i) = testsuite(j') - testsuite(i')$ and
3. the only source files changed by developers to reach version j were `.c` (C source), `.h` (C header or interface), `.y` (C parser source) or `.l` (C lexer source)

The second condition requires a minimal $|i - j|$. The set of positive tests is defined as $testsuite(i) \cap testsuite(j)$. The negative tests are $testsuite(j) - testsuite(i)$. Note that the positive and negative tests are disjoint.

Given a viable candidate subject program, its most recent test suite, and a range of viable revisions, we construct a set of testable bugs by considering each viable version i and finding the minimal viable version j, if any, such that there is a testable bug between i and j. We consider all viable revisions appearing before our start date in late May, 2011 as a potential source of testable bugs. However, we capped each subject program at 45 defects to prevent any one program from dominating the results.

Given these criteria, we canvassed the following sources:

1. the top 20 C-foundry programs on `Sourceforge.net`
2. the top 20 C programs on Google code
3. the largest 20 non-kernel Fedora 13 source packages

Program	Defects Repaired	Cost per Non-Repair Hours	Cost per Non-Repair US$	Cost Per Repair Hours	Cost Per Repair US$
`fbc`	1 / 3	8.52	5.56	6.52	4.08
`gmp`	1 / 2	9.93	6.61	1.60	0.44
`gzip`	1 / 5	5.11	3.04	1.41	0.30
`libtiff`	17 / 24	7.81	5.04	1.05	0.04
`lighttpd`	5 / 9	10.79	7.25	1.34	0.25
`php`	28 / 44	13.00	8.80	1.84	0.62
`python`	1 / 11	13.00	8.80	1.22	0.16
`wireshark`	1 / 7	13.00	8.80	1.23	0.17
total	**55 / 105**	**11.22h**		**1.60h**	

Table 5.2: Repair results: 55 of the 105 defects (52%) were repaired successfully and are reported under the "Cost per Repair" columns. The remaining 50 are reported under the "Non-Repair"s columns. "Hours" columns report the wall-clock time between the submission of the repair request and the response, including cloud-computing spot instance delays. "US$" columns reports the total cost of cloud-computing CPU time and I/O. The total cost of generating the results in this table was $403.

4. programs in other repair papers [201, 215] or known to the author to have large test suites

Many otherwise-popular projects failed to meet these criteria. Many open-source programs have nonexistent or weak test suites; opaque testing paradigms; non-automated GUI testing; or are difficult to modularize, build and reproduce on our architecture (e.g., `eclipse`, `firefox`, `ghostscript`, `handbrake`, `openjpeg`, `openoffice`). For several programs, we were unable to identify any viable defects according to our definition (e.g., `gnucash`, `openssl`). Some projects (e.g., `bash`, `cvs`, `openssh`) have inaccessible or unusably small version control histories. Other projects were ruled out by our test suite time bound (e.g., `gcc`, `glibc`, `subversion`). Some projects have many revisions but few viable versions that compile and run against recent test cases (e.g., `valgrind`). Earlier versions of certain programs (e.g,. `gmp`) require incompatible versions of `automake` and `libtool`.

The full set of benchmark programs and defects appears in Table 5.1. We acknowledge that it is not complete and that other additions are possible. However, while it is certainly "best effort," to our knowledge it also enables the most systematic evaluation of automated program repair to date. We have released this dataset publicly for the use of the software engineering research community.

5.4 How many defects can GenProg repair, and at what cost?

This section presents the results of running GenProg on each of the 105 bugs in the ManyBugs defect set in Table 5.1. This experiment addresses several of our empirical claims. First, it allows us to directly test Hypothesis 2 by measuring the percentage of human-repaired defects GenProg can repair. Second, it provides evidence to substantiate the claim tested by Hypothesis 3 by applying GenProg to real-world systems of indicative size, with test suites of indicative

size. The benchmarks in Table 5.1 include 5.1 million lines of code from 8 subject programs, and real-world test suites that include from tens (`gzip`) to thousands (`php`) of test cases. Because we ran these experiments on the Amazon EC2 framework, we can also report direct, publicly-comparable time and economic costs of using the GenProg technique, which in turn enables a comparison between the cost of running GenProg to repair a bug and the cost of asking a human to do the same.

5.4.1 Repair Results

Table 5.2 reports results. GenProg successfully repaired 55 of the defects (52%), including at least one defect for each subject program. The 50 "Non-Repairs" met time or generation limits before a repair was discovered. We report costs in terms of monetary cost and wall clock time from the start of the request to the final result, recalling that the process terminates as soon as one parallel search finds a repair. Results are reported for cloud computing spot instances, and thus include a one-hour start lag but lower CPU-hour costs. Table 5.2 does not include time to minimize the initial repairs because, as shown in Chapter 4 and Chapter 6, this step represents a small fraction of the overall cost, and is optional.

For example, consider the repaired `fbc` defect, where one of the ten parallel searches found a repair after 6.52 wall-clock hours. This corresponds to 5.52 hours of cloud computing CPU time per instance. The total cost for the entire bug repair effort for that to repair that defect is thus $10 \times 5.52 \text{ hours} \times \$0.074/\text{hour} = \$4.08$ (see Section 5.2).

The 55 successful repairs return a result in 1.6 hours each, on average. The 50 unsuccessful repairs required 11.22 hours each, on average. Unsuccessful repairs that reach the generation limit (as in the first five benchmarks) take less than 12+1 hours. The total cost for all 105 attempted repairs is $403, or $7.32 per successful run. These costs could be traded off in various ways. For example, an organization that valued speed over monetary cost could use on-demand cloud instances, reducing the average time per repair by 60 minutes to 36 minutes, but increasing the average cost per successful run from $7.32 to $18.30.

We view the successful repair of 55 of 105 defects from programs totaling 5.1 million lines of code as a very strong result for the power of automated program repair. This experiments provide evidence that GenProg can repair at least 50% of the defects that human developers fix in practice, as it demonstrably did so for an indicative set of defects (directly supporting Hypothesis 3). This completes our evaluation of GenProg's expressive power.

These results additionally demonstrate that GenProg can fix bugs in programs of up to several million lines of code with up to several thousand test cases, which are sizes consistent with many systems in practice. This addresses the first concern expressed in Hypothesis 3. We also view an average per-repair monetary cost of $7.32 as a strong efficiency result; we analyze how this cost compares to human developer costs in the next subsection.

5.4.2 Cost comparison

The experiment in the previous subsection resulted in a per-bug repair cost of $7.32, factoring in the cost of non-repair: The entire table cost $403 to generate using August, 2011 prices for Amazon's EC2 compute framework. These costs are reproducible, and we have released the experimental setup and virtual machine images for the use of the general research community. We argue that this monetary cost, and the associated time cost of 96 minutes per actual repair, provides strong evidence that GenProg is meeting its design goal of being competitive with human developers.

Although it is difficult to provide direct cost comparisons to the human-generated repairs for the same defects, because our benchmark programs do not report per-issue effort tracking, several indirect measures of time and monetary costs support this claim. As an indirect time comparison, Weiß *et al.* [4] survey 567 effort-tracked issues in `jboss` (an open-source Java project of comparable scale to our subject programs). Their mean time taken per issue was 15.3 hours with a median of 5.0 hours, with issues in the top half of the severity scale taking 2.8 to 5.8 times longer than others. Reports also suggest that it takes human developers 28 days on average to address even security-critical repairs [6], and nine days elapsed between the posted exploit source for the `wu-ftpd` format string vulnerability repaired in Chapter 4 and the availability of its patch.

As an indirect cost comparison, the Tarsnap.com bug bounty averaged $21 for each non-trivial repair (Section 5.1). Similarly, an IBM report gives an average defect cost of $25 during coding (rising to $100 at build time, $450 during testing/QA, and $16,000 post-release) [5, p.2]. In personal communication, Robert O'Callahan of Novell, the lead engineer for the Mozilla Gecko layout engine, noted that our costs would be "incredibly cheap if it carried over to our project!" but cautioned that the fix must be the right one to avoid damaging the long-term health of the code.

We note three potential complexities in these cost comparisons. First, we require test cases that identify the defects. This is standard practice in some organizations (e.g., at IBM, testing/QA might prepare test cases for particular bugs that separate maintenance developers may then use to fix them). In others (e.g., much open-source development), test case construction may introduce additional costs. Therefore, the $21–$25 cost of a bug fix reported by other practitioners may not be directly comparable to the $7.32 figure reported here, as generating the test case with the failing input (such as from a crash or defect report) will add overhead. However, even if test case construction triples or quadruples the cost of patch generation in this application, our costs remain competitive. Second, candidate patches produced by our technique will typically be inspected and validated by developers. While even incorrect tool-generated patches have been shown to reduce the amount of time it takes developers to address an issue [145], the exact reduction amount is unknown. Finally, we again note that human patches are imperfect: in a survey of 2,000 OS fixes, Yin *et al.* find that 14–24% are incorrect and 43% of those bad fixes led to crashes, hangs, corruption, or security problems [8]. As a result, directly comparing to the reported costs of issues management by humans in practice is difficult, because those costs do not differentiate between issues that are resolved adequately the first time and those that must be reopened or

Program	Repaired Default	Repaired w/ Annotation	Time Default	Time w/ Annotation	Unique Patches	Patches Per Repair
`fbc`	1 / 3	2 / 3	6.52	1.05	1	1.0
`gmp`	1 / 2	2 / 2	1.60	1.07	2	2.0
`gzip`	1 / 5	4 / 5	1.41	1.01	8	8.0
`libtiff`	17 / 24	19 / 24	1.05	1.03	115	6.8
`lighttpd`	5 / 9	6 / 9	1.34	1.24	23	4.6
`php`	28 / 44	33 / 44	1.84	1.20	157	5.6
`python`	1 / 11	2 / 11	1.22	1.16	5	5.0
`wireshark`	1 / 7	3 / 7	1.23	5.02	7	7.0
total	**55 / 105**	**71 / 105**	**1.60h**	**1.35**	**318**	**5.8**

Table 5.3: Alternate defect repair results. The first set of columns compares success rate and time to repair between our default repair scenario and a scenario that includes human localization annotations. "Unique Patches" counts the number of distinct post-minimization patches produced if each of the 10 (unannotated) parallel searches is allowed to run to completion.

readdressed later.

One of the `php` defects that GenProg successfully repairs corresponds to a use-after-free vulnerability with an associated security CVE.[8] The human patch uses intermediate variables to hold deep copies of the function arguments such that when one is destroyed, the others are unaffected. GenProg inserts code that copies the vulnerable argument an additional time, preserving the relevant values when they are converted. We note that all three of the bug bounties mentioned in Section 5.1 pay at least $500 for a single security fix, which exceeds the entire cost of our 105 runs ($403) — including the one that obtained this security repair.

We thus argue that the results in this chapter provide evidence to support our claim that GenProg can repair real bugs in real programs (Hypothesis 2, addressing expressive power) at a cost that is human competitive (Hypothesis 3, addressing scalability).

5.5 What is the impact of alternative repair strategies?

The previous section tested our hypotheses regarding GenProg's expressive power (in terms of the proportion of real-world defects it can repair) and scalability (in terms of the sizes of the programs and test suites to which it applies, and the costs of doing so). Although the results increase confidence that GenProg is effective and cheap, an important threat to the validity of those results is overfitting of the algorithm and its use case. Put differently, the results in the previous section only suggest that there is *one way* to use GenProg that has produced good results. It is worthwhile to investigate alternative applications of the technique, including alternative parameter and operator choices and alternative use cases. We investigate algorithmic representation and operator choices in detail in Chapter 6; in this section, we

[8] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1148

evaluate alternative use cases for the algorithm. In particular, we investigate two alternative repair strategies: using annotations, and searching for multiple repairs.

5.5.1 Include human annotations

The evaluation in previous subsections evaluates GenProg in a fully automated context. However, researchers have demonstrated the feasibility of human-guided program synthesis, wherein developers provide a high-level outline or "sketch" for a program structure, and are guided via an automatic tool to generate the rest of the program code correctly (Chapter 2 provides a high-level discussion of template-based program synthesis). In sketching, a programmer specifies high-level implementation strategies—an outline of general structure, as well as details such as likely relevant variables, invariants, or function calls—but leaves low-level details to a program synthesizer. The synthesizer uses these inputs to generate the complete code. A feasible alternative use case for GenProg incorporates programmer annotations to guide the repair search, similar in spirit to program synthesis via sketching [216].

In this subsection, we evaluate GenProg's expressive power and human-comparable costs in this alternative application. In these experiments, we relax our assumption of full automation, and assume that humans provide an unordered superset of statements that may be used to construct a patch (i.e., fix localization information) and pinpoint critical areas where patch actions mights be applied (i.e., fault localization). Such annotations are easier to provide than a concrete patch [216], but are not automatic. We are interested in annotations to explore the upper limits of our fully automated method and to explore what a hybrid human-machine approach might achieve. We use the actual human repairs for our defect set as the source of our annotations. We say that a defect can be *repaired with annotations* if (1) it can be repaired automatically, or (2) it can be repaired with fault and fix information restricted to those lines and changes made by the human developers.

The first four columns of Table 5.3 show results. With annotations, statement-level repair patches 71 out of 105 bugs (68%), as compared to 55 out of 105 without annotations, and time to first repair decreases on average. If we break down the time to repair by scenario (instead of by benchmark, as shown in the table), approximately 50% of the scenarios are repaired more quickly with annotations, 30% are repaired more quickly without annotations, and 20% are repaired approximately equally quickly between the two approaches.

Manual inspection of the results suggest that annotations often improve localization across both the fault and the fix spaces: the hypothetical human annotator has pinpointed where the change should be inserted and provided candidate code to use in the insertion. Indeed, improved fault localization appears to contribute strongly to the speedup that annotations provide over the default localization. In several cases, a large reduction in the fault space appears to be the primary reason GenProg can repair a given annotated scenario but not its unannotated counterpart; we investigate this hypothesis in more detail in Section 6.1.2.

In the majority of cases, however, improved fix localization—that is, a smaller or more precise set of choices for candidate repairs—appears to be the primary reason that annotations lead GenProg to repair additional bugs. For example, a human developer repaired a bug in `gzip` simply by inserting `ifd = fileno(stdin);`. Although the location at which this code is inserted is found in the fault space produced with the default localization, the exact inserted code does not appear anywhere else in the program (cf. the GenProg design choice in Section 3.5.3). In order to repair the same bug without annotations, GenProg has to identify a combination of other statements that accomplish the same effect, which it did not do within 12 hours in the initial experiment. Restricting inserted code to that found elsewhere in the same program appears sufficiently expressive for many (more than 50%) of the bugs that humans address in practice. However, these results suggest that there are instances where this restriction reduces GenProg's ability to find suitable repair code in a reasonable amount of time. In these instances, additional information about reasonable candidate repairs may be necessary to achieve sufficient expressive power.

The annotations in this experiment serve as a theoretical upper bound on how much a human programmer can help GenProg in the search for repairs: the annotations we use are taken from "known good" answers, according to historical data. These results suggest that such annotations could provide a measurable improvement in GenProg runtime and increase expressive power, though the benefits would likely be less than shown here in practice, since real-world annotators would likely be less precise. The cost savings gained from more precise localization are partially counterbalanced by the cost of including a human in the process. Thus, we expect that the primary benefit of human annotations would be to improve expressive power, allowing GenProg to repair certain bugs that it could not repair without additional guidance regarding suitable fix code.

Overall, these results remain consistent with our claim that GenProg can be used to repair defects in an expressive and cost-effective way. They are also consistent with the relationship between search space size and repair success that we identify in the next chapter (Chapter 6) and suggest that benefits might be gained from improved localization, provided either by a human developer, existing research in fault localization, or future improvements in fix localization.

5.5.2 Search for multiple repairs

Diverse solutions to the same problem may provide multiple options to developers [145], or enable consideration of multiple attack surfaces in a security context [28]. The latter application could serve as a natural extension of the closed-loop detection and repair system we proposed and investigated in Chapter 4.

To investigate GenProg's utility in generating multiple repairs in terms of both effectiveness (expressive power) and cost (again, temporal and monetary), we allowed each of the ten independent trials per bug to run to completion instead of terminating early when any trial found a repair. To identify unique patches, we minimize the initial repairs to final

repairs using the combination of DIFFX [204] and delta debugging [202] described in Section 3.7. We consider a repair unique if the minimized patch is textually unique.

The last two columns in Table 5.3 shows how many different patches were discovered in this use-case. GenProg produced 318 unique patches for 55 repairs, or an average of 5.8 distinct patches per repaired bug. The unique patches are typically similar, often involving different formulations of guards for inserted blocks or different computations of required values. Because all trials, including successful ones, must now run to completion, the total cost increases from $403 to $502 for all 550 runs.

This experiment provides evidence that GenProg is expressively useful and that its costs remain human competitive in an alternative application, beyond the use case evaluated in Section 5.4. In this experiment, cost increased from approximately $7 per repaired defect to approximately $10 (resulting in multiple candidate repairs, which in turn provides additional utility), and time to generate all candidate repairs increases as well over the 12 hour period. These results remain competitive with human developer costs as discussed in Section 5.4.2.

5.6 How do automated and human-written repairs compare?

In this subsection, we compare the repairs produced by humans with those produced by GenProg for two indicative defects from the larger set. The goal of this subsection is to provide additional insight regarding repair quality, similar to the descriptions provided in Chapter 4. These case studies provide additional an qualitative test of Hypothesis 4, which concerns itself with the quality of GenProg patches.

5.6.1 Python Date Handling

In one bug, six **`python`** tests failed based on whether the date "70" maps to "1970" or "70". The human patch removed a global dictionary, 17 lines of processing using that dictionary, and a flag preserving that dictionary during **`y2k`** checking. The automated repair removes the 17 lines of special processing but leaves untouched the empty dictionary and unused flag. This retains required functionality but increases run time memory usage by one empty dictionary. Even if our approach had considered removing the dictionary declaration (which is not possible with our statement-level operations), that removal would have been dropped from the candidate patch during minimization because it is not necessary to pass all tests. The patch is thus as functionally correct as the human patch but degrades some non-functional aspects (maintainability and memory footprint), neither of which are tested.

This "normal" priority issue[9] was open for 7 days and involved 12 developer messages and two different candidate patches submitted for review by human developers.

[9] http://bugs.python.org/issue11930

5.6.2 Php Global Object Accessor Crash

`php` uses reference counting to determine when dynamic objects should be freed. `php` also allows user programs to overload internal accessor functions to specify behavior when undefined class fields are accessed. Version 5.2.17 had a bug related to a combination of those features. At a high level, the "read property" function, which handles accessors, always calls a deep reference count decrement on one of its arguments, potentially freeing both that reference and the memory it points to. This is the correct behavior *unless* that argument points to `$this` when `$this` references a global variable—a situation that arises if the user program overrides the internal accessor to return `$this`. In such circumstances, the global variable has its reference count decremented to zero and its memory is mistakenly freed while it is still reachable, causing the interpreter to incorrectly return an error later.

The human-written patch replaces the single line that always calls the deep decrement with a simple if-then-else: in the normal case (i.e., the argument is not a class object), call the deep decrement on it as before, otherwise call a separate shallow decrement function (`Z_DELREF_P`) on it. The shallow decrement function may free that particular pointer, but not the object to which it points.

The GenProg patch adapts code from a nearby "unset property" function. The deep decrement is unchanged, but additional code is inserted to check for the abnormal case. In the abnormal case, the reference count is deeply incremented (through machinations involving a new variable) and then the same shallow decrement (`Z_DELREF_P`) is called.

Thus, at a very high level, the human patch changes `deep_Decr()` to:

```
1 if (normal)
2   deep_Decr();
3 else
4   shallow_Decr();
```

while the GP-generated patch changes it to:

```
1 deep_Decr();
2 if (abnormal) {
3   deep_Incr();
4   shallow_Decr();
5 }
```

The logical effect is the same but the command ordering is not. After formatting, both patches increase the file size by four lines. The human patch is perhaps more natural: it avoids the deep decrement rather than performing it and then undoing it.

5.7 Discussion and Summary

In this chapter, we developed an experimental framework and benchmark set to test two hypotheses about GenProg's expressive power and scalability. Specifically, we measured the proportion of real-world bugs it can repair and how competitive it is as compared to human developers, in terms of time and monetary cost. The ManyBugs benchmark set consists of 105 reproducible defects that developers have previously patched. Those defects come from 8 programs including 5.1 million lines of code and 10,193 test cases. This evaluation includes orders of magnitude more code, test cases, and defects than related or previous work [13, 171, 181, 217].

GenProg repaired 55 of 105 defects from the programs in the ManyBugs dataset in 1.6 hours for $7.32 each, on average; the entire table in Section 5.4 can be reproduced for $403. These costs includes a 1-hour start time; paying more for on-demand instances reduces trial time, but increases cost. All defects in the ManyBugs dataset are at least moderately severe and were important enough for developers to fix. Additional qualitative comparison suggests that GenProg's repairs are often functionally equivalent to the human patches, but deemphasize untested non-functional requirements such as memory usage, readability or maintainability.

Our extensive use of parallelism is novel compared to previous work (cf. [217, Sec 3.4]) and yields an average return time of 96 minutes per successful repair. However, if we continue to search beyond the first repair, GenProg finds 5.8 unique patches per successful repair, which provides developers more freedom and information. When augmented with sketching-inspired annotations [216], GenProg repairs 71 of the 105 defects. The remaining 34 presumably require algorithmic or localization improvements. All three candidate use cases for the algorithm result in time and monetary costs that are competitive with those associated with human bug repair.

These results give us confidence both that GenProg can scale and apply to the real-world bugs that we are targeting in our design, and that its costs are viable for that setting as well. These results serve to support the claims formalized by the hypotheses we tested in this chapter, first presented in Chapter 1. GenProg repaired more than 50% of an indicative set of bugs in large systems with variable test suite sizes. Humans previously addressed those same bugs and wrote test cases that checked for the associated behavior, suggesting that they correspond to real-world concerns. Finally, the cost of doing so appears viable as measured against the cost of human repairs.

An important threat to validity involves whether our results generalize to other settings (i.e., whether our benchmarks represent an indicative sample). We attempt to mitigate selection bias (i.e., "cherry-picking") by defining viable subject programs and defects and then including all matching defects found by an exhaustive search. We acknowledge that our benchmark set is "best effort," however. Our requirements limit the scope from which we draw conclusions. For example, using deterministic bugs leaves race conditions out of scope, while using only C code leaves multi-language bugs out of scope. In addition, we only evaluate on open source software, and thus cannot directly speak to industrial development. However, using checked-in bugs that trigger checked-in test cases has the advantage that all bugs

considered were of at least moderate priority. This experimental choice serves to mitigate this threat to validity.

The results in this chapter substantiate our claims regarding GenProg's expressive power and scalability. However, we have not discussed in detail the factors that influence scalability and performance. The benchmark suites we have developed for the evaluation of automatic program repair enable additional investigations along these lines, as well as of representation and operator choices (which have been shown to matter more than parameter values in this setting [218]). The next chapter addresses these concerns.

Chapter 6

Representation, operators, and the program repair search space

THE results in the preceding chapters help substantiate our claims that GenProg produces patches of sufficient quality for practical use, that it is expressive and explicitly human competitive, and that it scales to real-world systems. We have shown that it can repair many types of bugs in many different programs, and that it can repair more than half of an indicative set of bugs that human developers repaired in practice.

Natural next questions concern *how* and *why* GenProg works, what factors contribute to the empirical behavior presented so far, and how and whether GenProg can be improved. We designed GenProg to take advantage of the special structure of the automatic repair search problem, and we use domain-specific knowledge to constrain the search space intelligently. We attribute at least some of GenProg's success to the choices of representation, fitness function, genetic operators, and effective search space constraints. However, in those respects, and many others, the algorithm that we have developed for GenProg differs significantly from previous work in genetic programming. This can be seen by examining surface-level features of the algorithms presented in state-of-the-art research in GP [155]. Many applications of GP developed within the lasts five years operate on different search spaces, such as expression trees, and tend to run for many more iterations with much larger populations than does GenProg [189]. The design of our novel genetic programming algorithm for automated program repair is a key contribution of this thesis; the nature of the search algorithm therefore warrants especial consideration.

Additionally, there has been little previous work in this domain that explicitly compares or evaluates representations, operators or the effective search spaces, although a number of options have been proposed. For example, the algorithm presented and evaluated throughout this dissertation represents intermediate variants as lists of edits to an initial program [164, 219] (which we refer to as the PATCH representation throughout this chapter). In previous preliminary

work, we proposed to represent candidate solutions as full abstract syntax trees with a weighted list of statements to mutate [217] (referred to in this chapter as AST/WP). Other researchers have represented variants as lists of bytecode [168] or assembly statements [163]. Mutation operators range from highly specialized (e.g., changing relational operators or variable names [164]); to pre-specified by the GP engine based on language-specific primitives (e.g., [169]); to grammar-based rewrite rules [220]; to generic insertions, replacements, or deletions of statements in an AST [217]. Certain applications demand a prominent treatment of particular operators (e.g., a semantics-preserving crossover operator for Java bytecode [168]). Popular crossover choices include crossback [169, 217], one-point [163, 164], and variations of uniform crossover [219]. Parameter choices vary, and tend to be selected heuristically or by deferring to previous work. By and large, Arcuri *et al.*'s [169] summary is characteristic: "In general, we set the parameters with values that are common in literature . . . the ones reported are the best we found."

The goal of this chapter is therefore to analyze in detail the design decisions that underlie the GenProg algorithm. The approach we take is two-fold. We begin by expanding on the results in Chapter 4 and Chapter 5, focusing specifically on what factors influence performance and scalability. We evaluate algorithmic performance in two ways: time to repair (Section 6.1) and GP success (Section 6.2).

In terms of repair time, our analyses suggest that the most time-intensive component of the algorithm is fitness evaluation (Section 6.1.1), during which intermediate variants are run on the test suite; varying the number of test cases explores the tension between correctness and execution time, which we investigate with a case study (Section 6.1.1). Because test suite execution dominates repair time, we use the number of fitness evaluations to a repair as a program-independent performance metric. This allows us to investigate scaling behavior; recall that one design goal was an algorithm that scales sublinearly with program size. Performance in these terms scales in time proportional to the size of the search space (Section 6.1.2), which is more strongly related to fault localization accuracy than it is to program size.

Looking at algorithmic success on any given bug, we investigate whether GenProg appears to only repair "unimportant" bugs across a number of metrics, but cannot support the concern statistically (Section 6.2.1). Instead, the localization and search space characterization appear to play a larger role in whether GenProg can repair a given bug than do external measures of bug importance (Section 6.2.2).

We conclude from these analyses that GenProg success and performance both in general and on any particular bug are strongly (though not exclusively) influenced by internal algorithmic components such as the representation and operator choices. This is consistent with the conclusions other researchers have drawn [218].

Thus, in the second half of this chapter, we conduct a an in-depth study of several critical representation, operator and other design choices used for evolutionary program repair at the source code level; we outline the components under study in Section 6.3. We modify GenProg's internals and evaluate its success on the ManyBugs benchmark set proposed in Chapter 5. These experiments empirically evaluate the design choices embodied in the algorithm, and

enable normative recommendations for the improvement of search-based automatic program repair. Specifically, we provide:

- A direct comparison of two source-level representations used for automatic patch generation at the source code level, and an analysis of which representation features contribute most to success rate (Section 6.4.2).
- An empirical evaluation of competing crossover operators as applied to the PATCH representation we have proposed and evaluated throughout this dissertation (Section 6.3.2).
- An empirical study of the role of mutation operators in successful repairs and the effect of varying the probabilities with which different operators are applied on both GP success rate and time to repair (Section 6.3.3).
- An evaluation and analysis of how best to characterize the fault space, and specifically the validity of our assumption so far that genetic modifications should focus on statements executed exclusively by buggy inputs (Section 6.3.4).

Overall, we find that these choices do individually matter, especially for the more difficult repair scenarios in terms of both repair success and time, and that they combine synergistically in their contributions to GenProg's overall effectiveness.

6.1 What influences repair time?

This section analyzes factors that contribute to and influence GenProg execution time in the experiments from the preceding two chapters. Time to repair can be measured in two ways. First, we measure the wall clock time that a user waits for GenProg to complete on a given bug (in seconds). Second, we measure the number of fitness evaluations to a repair. In this section, we first discuss factors that contribute to wall-clock time, and the tradeoffs those factors present (Section 6.1.1), and then discuss the scaling behavior of the algorithm as measured by the number of fitness evaluations required to find a repair (Section 6.1.2).

6.1.1 Test suite size and wall-clock repair time

We begin by examining the repair results on the benchmarks from in Chapter 4, Table 4.1. Although their test suites vary somewhat in size, they were hand-modified and in some cases hand-generated (i.e., the test suites and the programs were developed separately), and thus these benchmarks allow us to examine the components of GenProg runtime independently from the choices program-specific developers made about testing.

Program	All Tests	Positive Tests	Negative Tests	Compilation
`gcd`	91.2%	44.4%	46.8%	8.4%
`zune`	94.7%	23.2%	71.5%	3.7%
`uniq`	71.0%	17.2%	53.8%	25%
`look-u`	76.4%	17.1%	59.3%	20.7%
`look-s`	83.8%	29.9%	53.9%	12.9%
`units`	57.8%	20.7%	37.1%	35.5%
`deroff`	44.5%	8.9%	35.6%	46.5%
`nullhttpd`	74.9%	22.9%	52.0%	12.8%
`openldap`	93.7%	82.6%	11.1%	6.2%
`indent`	36.5%	16.4%	20.1%	43.1%
`ccrypt`	87.3%	65.2%	22.0%	4.7%
`lighttpd`	68.0%	67.9%	0.06%	25.3%
`flex`	23.2%	18.3%	4.8%	39.5%
`atris`	1.9%	0.8%	1.1%	64.9%
`php`	12.0%	2.0%	10.0%	78.4%
`wu-ftpd`	87.2%	38.6%	48.6%	6.6%
Average	62.75%	29.76%	32.99%	27.14%
StdDev	30.37%	24.00%	23.17%	22.55%

Table 6.1: Percentage of total repair time spent on particular tasks. In general, fitness evaluation (i.e., running tests) dominates GenProg execution time.

Components of GenProg wall-clock time

Running test cases to evaluate variant fitness is the dominant cost associated with GenProg. Table 6.1 shows the proportion of time taken by each important algorithmic component. An average repair run for the benchmarks in Table 4.1 took 356.5 seconds.Executing the test cases for the fitness function takes much of this time: on average, positive test cases take $29.76\% \pm 24.0$ and negative test cases $32.99\% \pm 23.17$ of the time. In total, fitness evaluations comprise $62.75\% \pm 30.37$ of total repair time. Many test cases include timeouts (e.g., negative test cases that specify an infinite-loop error); others involve explicit internal delays (e.g., ad hoc instructions to wait two seconds for the web server to get "up and running" before requests are sent; the `openldap` test suite makes extensive use of this type of delay), contributing to their runtime. Compilation of variants averaged $27.13\% \pm 22.55$ of repair time. Our initial implementation makes no attempt at incremental compilation, which would reduce this cost. The high standard deviations arise from the widely varying test suite execution times (e.g., from 0.2 seconds for `zune` to 62.7 seconds for `openldap`), even on these hand-selected examples.

The test cases comprise fitness evaluation and define patch correctness; test suite selection is thus important to both scalability and correctness. We saw several examples of this tradeoff in Chapter 4. For example, on the `nullhttpd` benchmark, omitting a positive test case that exercises POST-functionality led to a repair that removes that functionality. Adding positive test cases can actually improve fault localization by increasing the coverage of those test cases relative to the statements executed by the negative test cases. This can reduce the search space, but increase run time. Thus,

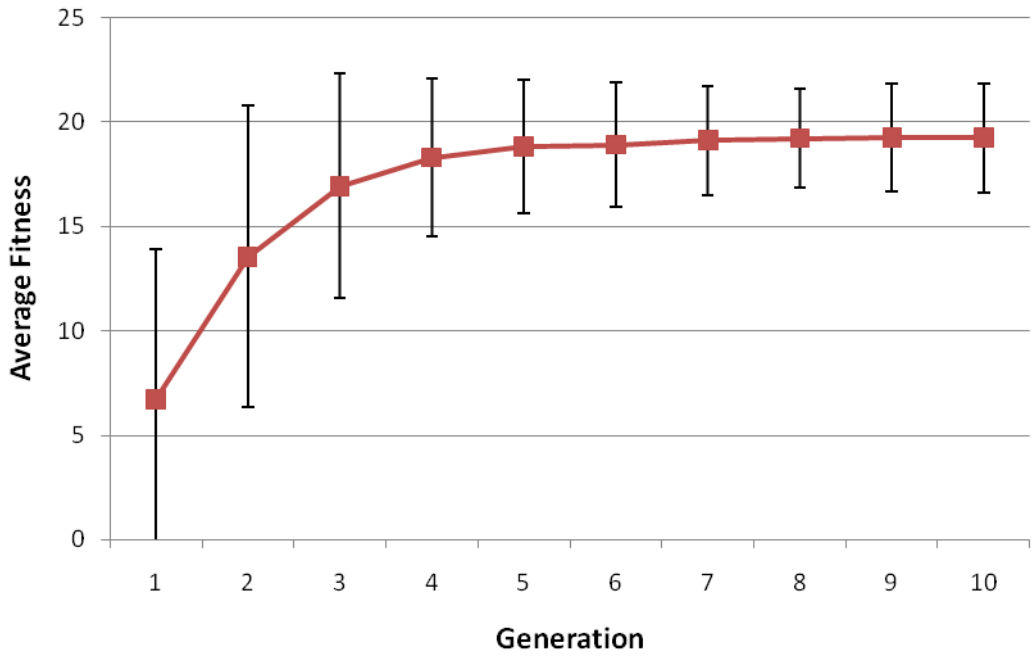

Figure 6.1: Evolution of the Zune bug repair with 20 positive test cases and 4 negative test cases, all equally weighted. The boxes represented the average over 70 distinct trials; the error bars represent one standard deviation.

additional test cases can not only protect core functionality but also improve the success rate. The next subsection presents a case study that explores the effects of varying test suite size.

Varying the Number of Test Cases

The benchmarks in Chapter 4 are chosen for their variety, and thus we limited the size of their test suites to separate scalability from expressive power as concerns. In this subsection, we investigate the effects of increasing the size of the test suite for one of those benchmarks; in effect, we examine the cross-section of the expressivity concerns of Chapter 4 and the scalability concerns of Chapter 5 by evaluating the effects of a controlled increase in test suite size on the Zune bug, which GenProg initially repaired using 6 test cases.

Figure 6.1 shows the averaged results of 70 trials on the Zune bug, using a fitness function with 24 test cases: 20 positive test cases and 4 negative test cases. The error bars represent one standard deviation. For simplicity of presentation, all test cases are shown weighted equally. Unsurprisingly for a search algorithm, early generations have fitness values with high variance, and in later generations the variance decreases as the average fitness increases. The original program passes the positive test cases but fails the negative test cases; it thus has a fitness of 20. Note that over all generations, the average fitness is below the baseline of 20, indicating that the majority of individuals are worse than the original program. The baseline value of 20 represents a plateau that must be exceeded; obtaining the final repair requires descending from that local maximum to climb to the global maximum. Thus, the primary repair is discovered

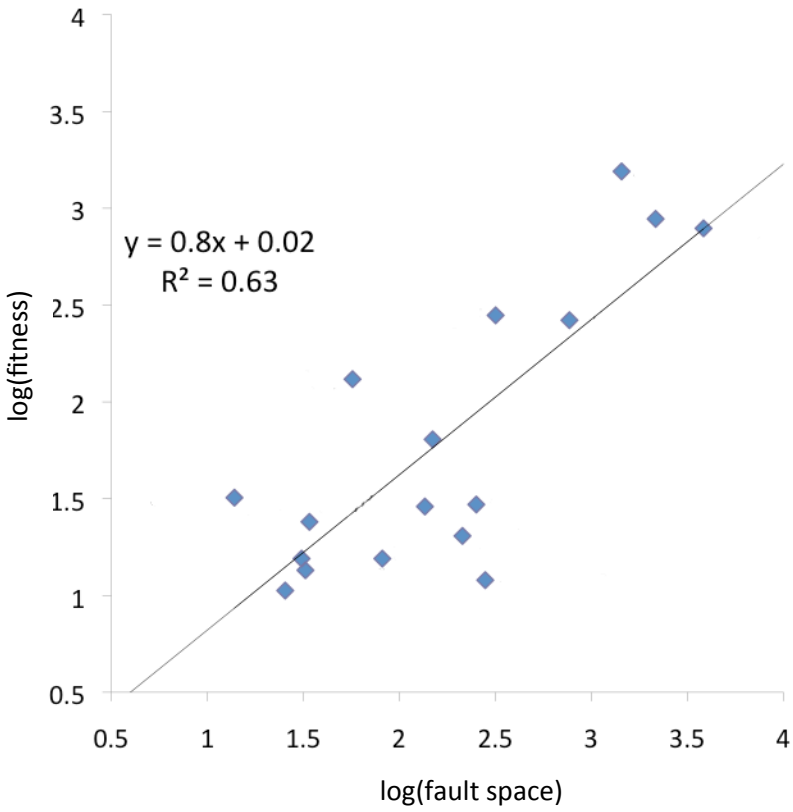

Figure 6.2: GenProg execution time scales with fault space size. Data shown for 17 benchmark programs used in expressive power evaluations in Chapter 4 and [201]. The x-axis shows fault space size (sum of all fault-localization weights over all program statements); the y-axis shows the number of fitness evaluations performed before a primary repair is found (averaged over 100 runs). Note the base-10 log-log scale.

by first losing fitness and then regaining it on the way to the global optimum.

Intuitively, additional test cases could reduce success rate by overly constraining the search space. However, the opposite happened in this example. Using seven test cases, the average success rate is 72%, while the average success rate using 24 test cases is 75%. However, adding test cases does dramatically increase the total running time of the algorithm: with seven test cases, the average time to discover the primary repair is 56.1 seconds; with 24, this time increases to 641.0 seconds. This motivates our use of $SampleFit$ on intermediate variants in lieu of $FullFitness$ (see Section 3.6) for the larger benchmarks in Chapter 5. However, assuming that a repair can be found, scalability favors a fitness function with a small number of fast-running test cases, because the evaluation of intermediate variants dominates execution time. This suggests that in addition to test suite sampling, test suite prioritization (see Chapter 2) might be profitably applied in this application to alleviate the trade off between runtime and correctness.

6.1.2 Time complexity and scaling behavior

We have empirically demonstrated that GenProg can repair bugs in large, real-world programs, which directly relates to our stated design goals of a human competitive repair approach, as well as Hypothesis 3, that GenProg scales to real-world system sizes. However, we have not yet investigated the factors that contribute to its scaling behavior. In this section, we examine the observed time complexity of GenProg as measured by the number of fitness evaluations to a repair, which allows us to aggregate observed behavior across many different programs.

We observe in the experiments in both Chapter 4 and Chapter 5 that GenProg execution time does *not* appear to correlate with program size. Indeed, we have statistically investigated this possible relationship and were unable to find a significant correlation or relationship. Instead, as we observed in Section 3.5, the space of candidate solutions that GenProg searches can be parametrized by the fault and fix localization of a given bug. We measure both fault and fix size by summing the fault and fix weights (respectively) of all statements in a program, providing a rough approximation to the size of each space. Note that fault space size in particular is based on observed test case behavior and not on the much larger number of loop-free paths in the program.

Figure 6.2 plots fault space size against the average number of fitness evaluations until the first repair, on a log-log scale, for the benchmarks from Table 4.1. The straight line suggests a relationship following a power law of the form $y = ax^b$ where b is the best-fit slope and $b = 1$ indicates a linear relationship. Figure 6.2 suggests that the relationship between fault space size search time is less than linear (slope 0.8). This observation is corroborated by the experiments on the ManyBugs benchmark set in Table 5.1; for that much larger set, we also found a statistically significant log-log relationship between fault space size and fitness evaluations to a repair ($r = 0.28$, $p = 0.01$).

Again on the ManyBugs benchmarks, we additionally find a significant *negative* correlation between the log of the fix space size and the log of the number of fitness evaluations required to find a repair ($r = -0.42$, $p < 0.0001$), suggesting that more options for a repair actually improves running time. One possible explanation for these results is that while bad fault localization can preclude a repair (e.g., the variable `x` must be zeroed just before *this* function call), imprecise fix localization may make it difficult but still possible (e.g., there are many ways to set `x` to `0` *without* using "`x=0;`"). A larger fix space may include more candidate repair options, reducing the time to find any one (although it does not appear to correlate with actual success; see Section 6.2).

The experiments in Section 5.5.1 provide additional support for these observations. For a number of scenarios, adding human annotations as hypothetical "templates" to guide a repair decreased GenProg repair time over the scenarios that used the traditional fault localization scheme. We hypothesized in the previous chapter that a non-trivial component of this performance improvement was the improved localization the annotations provided. We see this in a statistically significant correlation between GenProg speedup and the percent improvement in fault localization provided by the human annotators ($r = 0.84$, $p < 0.001$).

These relationships only approximate GenProg's scaling behavior, as fault and fix sizes are not the only measures of space complexity, and the relationships, while significant, are not especially strong. Thus, search time may not grow sub-linearly using other measures. However, these results are encouraging, because they suggest that GenProg scales at least partially with search space size as characterized by fault and fix localization, and, importnatly, *not* with program size. While fault space size is related to the accuracy of the fault localization for a given defect, fix space size does tend to grow with program size. It is thus especially encouraging that an increased fix space size appears to actually decrease repair time. These observations provide additional support for Hypothesis 3, that GenProg is scalable to large, real-world systems.

6.2 What influences success?

Given that GenProg appears able to handle bugs in large systems in a reasonable amount of time, this section explores factors that may influence GenProg's *success* in repairing a given defect. This analysis focuses on GenProg's repairs of the ManyBugs benchmark bugs from Chapter 5, because that set includes negative examples (i.e., bugs that were not repaired). As such, we can investigate questions regarding which internal or external factors influence GenProg's likelihood to repair a particular bug. Specifically, we investigate the relationship between GenProg success and defect complexity using several external metrics, including developer reported defect severity and the number of files touched by developers in a repair. We also consider internal metrics such as localization size.

6.2.1 Correlating Repair Success with External Metrics

One concern is that GenProg might succeed only on "unimportant" or "trivial" bugs. We investigate this hypothesis by analyzing the relationship between repair success and external metrics such as human time to repair, human repair size, and defect severity. With one exception, we were unable to identify significant correlations with these external metrics.

We manually inspected version control logs, bug databases, and associated history to link defects with bug reports. Although all of our benchmarks are associated with source control and bug-tracking databases, not all defect-associated revisions could be linked with a readily available bug report [214]. We identified publicly accessible bug or security vulnerability reports in 52 out of 105 of our cases. All bug reports linked to a defect in our benchmark set were eventually marked "confirmed" by developers; this helps support our claim that the ManyBugs defects are indicative of bugs that human developers considered important. We measure developer time as the difference between when the bug report was marked "assigned" and when it was "closed", which we know is a rough approximation. We extracted developer-reported defect severities on a 1–5 scale. For some projects priority ranking is almost mandatory (e.g., `python`), while in others it is optional (e.g., `php`). When possible, we used the ranking provided by the CVE database. We assigned `php` security bug reports marked "private" a severity of 4.5. Ultimately, we identified severity information

for 28 of the 105 defects. Results on this subset are comparable to those on the full dataset: GenProg repaired 26 of the 52 defects associated with bug reports (50%) and 13 of the 28 (46%) associated with severity ratings.

We investigated both linear and non-linear relationships between repair success and search time and the external metrics. We found a significant Pearson's correlation in only one case. The number of files touched by a human-generated patch is slightly negatively correlated with GenProg success ($r = -0.29$, $p = 0.007$): The more files the humans changed to address the defect, the slightly less likely GenProg was to find a repair We were unable to identify a significant relationship between either "human time to repair" or "human patch size (in `diff` lines)" and GenProg's repair success.

We found no significant correlation between "bug report severity" and "GenProg's ability to repair." Exploring further, we found no significant difference between the mean severity of repaired and unrepaired defects (Student T test and Wilcoxon Rank-Sum test) at $\alpha = 0.95$. These results suggest that the defects that GenProg can and those that it cannot repair are unlikely to differ in human-adjudged severity. We note that no defect in our benchmark set associated with a severity report has lower than "Normal" priority (3 in our scheme). Recall that, by construction, the ManyBugs dataset restricts attention to bugs important enough for developers to fix (see Section 5.3).

6.2.2 Correlating Repair Success with Internal Metrics

Returning again to the localization space, first discussed above (Section 6.1.2), we find a statistically significant, though not very strong, relationship between the log of the sum of the fault weights and repair success ($r = -0.36$, $p = 0.0008$). Combining this observation with those above, we observe that as fault space size increases, the probability of repair success decreases and the number of variants evaluated to a repair increases. We were unable to identify a statistically significant relationship between fix space size and repair success; the relationship between fix space size and repair time is discussed in Section 6.1.2.

These results are corroborated by the experiments in Section 5.5.1, in which we evaluated GenProg's utility when combined with human annotations. Manual inspection of the results suggests that in several cases, improved fault localization provided by the annotations contributed to GenProg's ability to repair programs on which it did not succeed without annotations. One bug in the `gmp` library stands out in particular as an interesting case study. GenProg failed to find a repair for this bug when run for 12 hours with the default localization strategy; with annotations, which decreased the size of the fault space by more than 90% and the fix space by more than 60%, GenProg succeeded in finding a repair within 1.60 hours. We note that, by default, the bug is poorly-localized, but that it appeared that GenProg should be able to repair it using the operators available to it. Indeed, GenProg *can* find a repair with the default localization strategy when run for much longer (almost two days). Similarly, localization annotations appeared to provide significant assistance to several `gzip` bugs, which were by and large very poorly localized by the default scheme, especially as

compared to the other benchmarks. These case studies illustrate the larger point, which is that in many cases, improved fault localization provided by a hypothetical human annotators appears to significantly contribute to search success. This suggests that more precise fault localization, such as adapted from related work in the field (Chapter 2 provides more detail), might provide significant improvements in the GenProg repair approach.

The human repairs for the bugs in the ManyBugs dataset also illuminate our decision to use only statement-level changes (first described in Section 3.4.1). Human developers used at least one "non-statement level" change (e.g., introducing a new global variable) in 33 of the 105 subject defects. The statement-level technique is less likely to repair such defects, addressing only 33% of them (vs. 52% overall repair rate). Statistically, whether a human repair restricts attention to statement-only changes moderately correlates with whether our technique can repair that same bug: $r = 0.38, p < 0.0001$. However, the unannotated statement-level approach can repair 11 of those defects. For example, we observed that humans often introduce new variables to hold intermediate computations or to refactor buggy code while repairing it. GenProg achieves the same effect (without the refactoring) by reusing existing variable definitions to hold intermediate results.

Restricting attention to statements reduces the search space by one to two orders of magnitude over a scheme that includes expressions, declarations, or definitions as viable sites for modification. The results presented so far suggest that the tradeoff is a good one. However, they also suggest that more powerful or finer-grained operators might allow GenProg to address other real-world defects that are currently outside its scope.

This analysis motivates a deeper look at the influence of GenProg's representation and operator choices and their effect on its performance; we address this question next.

6.3 Algorithmic components under study

The previous sections analyzed the results from Chapter 4 and Chapter 5 to identify the factors that appear to influence time to repair, scalability, and repair success. The analyses in those sections are unified by a common theme: given a program with a bug, GenProg success and runtime appear to be more strongly influenced by factors internal to the algorithm, such as search space characterization (Section 6.1.2 and Section 6.2.2) and operator choice (Section 6.2.2) than they are by external features such as bug severity or program size. As mentioned in the introduction to this chapter, previous work, including our own, has proposed a number of options for these internal operator and representation choices. However, to date there has been little examination of how these factors influence success in this domain, despite the atypicality of the space.

Therefore, the remainder of this chapter presents an in-depth study of four key areas of algorithmic design, and specifically how they impact repair success and runtime:

1. representation (Section 6.3.1)

2. crossover (Section 6.3.2)

3. mutation and selection (Section 6.3.3)

4. search space (Section 6.3.4)

We modify key areas of the GenProg algorithm and measure how those choices change its repair performance on our benchmark programs. The rest of this section outlines each of the areas of study in more detail, focusing on the experimental questions we evaluate. Section 6.4 gives empirical results.

6.3.1 Representation

First, we investigate the two best-established options for source-level automated program repair.

The *Abstract Syntax Tree/Weighted Path* (AST/WP) representation (e.g., [217]) defines a program variant as a pair consisting of its AST (with each statement uniquely numbered) and a *weighted path* through it, a simple form of fault localization typically defined by the statements executed by the failing test case. The weighted path is defined for the original program and remains unchanged during the search, even if introduced variations change control flow. Fitness is evaluated by pretty-printing the AST to produce source code, which is then compiled and run on the test cases. This representation frames the genetic programming for automatic repair problem in a more conventional way, as the search objective is a version of the program that is repaired, instead of a patch that repairs the input program.

The PATCH representation defines an individual as a sequence of edits to the original program source; it is the representation that GenProg uses by default in this dissertation. Ackling *et al.* [164] introduced a variant of this method, storing the edit list as a bitvector where each bit indexes an array of possible mutations to the underlying code. This is in contrast to our approach, which represents each patch as a variable-length sequence of edits to the statements of the original program's AST [219]. We study the latter representation for several reasons: (1) it admits a wider range of mutations, (2) it does not require pre-enumeration of all possible mutations (improving scalability), and (3) it is directly comparable to related work, especially the AST/WP representation in the context of the GenProg implementation.

We compare these two choices for source-level stochastic program repair to test the hypothesis implied in Section 3.4.1, that the PATCH representation provides a superior mechanism for traversing the space of candidate bug repairs. We also compare the use of *semantic check* and crossover to evaluate which high-level components of the representation and algorithm contribute the most to repair success.

6.3.2 Crossover

Program source representations support several crossover operators, only some of which we study here. For example, earlier work proposed a *crossback* operator [169, 217], in which intermediate variants underwent crossover with the

original program (rather than with another intermediate variant). Subsequent work found it equivalent to traditional one-point crossover [221] in our domain. We therefore restrict attention to heretofore unexamined crossover choices.

In the AST/WP representation, one-point crossover applies to statements on the Weighted Path. Given two parents p and q, a point along the weighted path is selected at random, and all statements after that point are swapped between the parents to produce two offspring. Only statements along the weighted path are affected.

The *patch subset* operator is a variant of uniform crossover for the PATCH representation [219]; it has been previously proposed as a candidate domain-specific operator specifically intended for automated patch evolution. This operator takes as input two parents p and q. The first (resp. second) offspring is created by appending p to q (resp. q to p) and then removing each element randomly with 50% probability. This operator allows edits to similar ranges of the program to be combined into one individual (e.g., parent p inserts B after A and parent q inserts C after A). This contrasts with AST/WP one-point crossover, where each offspring can receive only one edit to statement A.

The algorithm presented and evaluated in this dissertation makes use of one-point crossover applied directly to the patch representation [198], which selects crossover points p_n and q_m in parents p and q. The first half of p is appended to the second half of q, and vice versa, to create two offspring, as described in Chapter 3.

We evaluate these crossover operators by modifying the crossover operator and keeping all other algorithmic choices constant. We evaluate the results of new repair runs to draw conclusions about which crossover operator is most effective for automated repair.

6.3.3 Mutation

In both representations, mutation is restricted to AST nodes corresponding to C statements. A destination statement $stmt_d$ is chosen from the set of permitted statements according to a probability distribution (see Section 6.3.4). One of the available mutation operators is then selected (with equal probability, in previous chapters). The available mutation operators are **delete**, **insert**, and **replace** operators. In some work, **swap** is substituted for **replace**; **swap** on average is less successful than either **insert** or **delete**. If **insert** or **swap/replace** are selected, a second statement $stmt_s$ is also selected randomly to serve as the source for the candidate change code. These changes are either applied directly to the AST (in the AST/WP representation) or appended to the list of edits to be applied at fitness evaluation time (in the PATCH representation).

We investigate the distribution of the different mutation operators in the repairs presented so far, in which they were selected with equiprobability. Based on these observerations, we evaluate the effects of modifying the probability that each mutation is selected on repair success and time.

6.3.4 Search space

The search space of possible solutions in a given representation is infinite, but we have restricted the search to a smaller space that is likely to contain a repair. We parameterize these restrictions along three dimensions, as first presented in Chapter 3:

- **Fault space.** The search space is reduced by restricting mutations to program locations associated with incorrect behavior. These locations (i.e., program statements) are weighted to influence their probability of being mutated.
- **Mutation space.** The search space is further constrained by the set of mutations that are possible at each location and their selection probability. In previous chapters, mutations are selected with equal random probability.
- **Fix space.** In the case of an insertion or replacement mutation, a statement must be selected as the *source* to be copied. We refer to this as *fix localization* (first introduced in Section 3.5.3). Candidate fixes are restricted to those within the original program, and a *semantic check* eliminates the possibility of copying or moving statements that reference variables that will be out-of-scope in the new location.

These decisions constrain the mutation operators regardless of the representation choice. In AST/WP, the fault space is explicitly represented by the *weighted path*, which defines the sites of mutation operations and the basic unit (gene) used in crossover. The PATCH representation uses a similar weighting to guide selection of possible edit sites, although the weights are not stored explicitly with the individual variants.

We investigate the distribution of the *repair space* that resulted from running GenProg under our initial assumption about the appropriate weighting for W_{Path} (which prefers the mutation of statements exclusively executed by the failing test case). We look in particular at whether the statements modified in the initial and final repairs fall on the exclusively buggy path, or whether they fall on paths executed by both non-buggy and buggy inputs. We then investigate the effects of modifying this distribution accordingly.

6.4 Experiments

This section presents experimental results on four algorithmic and parameter choices for evolutionary program repair:

- **Representation:** Does AST/WP or PATCH give better results in terms of finding repairs, and which representation features contribute most to success? (Section 6.4.2)
- **Crossover:** Which crossover operator is best for evolutionary program repair? (Section 6.4.3)
- **Operators:** Which operators contribute the most to repair success, and how should the operators be selected? (Section 6.4.4)

Program	Fault	LOC	Tests
`gcd`	infinite loop	22	6
`uniq-utx`	segfault	1146	6
`look-utx`	segfault	1169	6
`look-svr`	infinite loop	1363	6
`units-svr`	segfault	1504	6
`deroff-utx`	segfault	2236	6
`nullhttpd`	buffer exploit	5575	7
`indent`	infinite loop	9906	6
`flex`	segfault	18775	6
`atris`	buffer exploit	21553	3
average		**6325**	**5.8**

Table 6.2: Benchmark C programs [217, Fig. 4], each with one defect and several human- or fuzz- generated test cases. Col. 1 gives the name of the program, Col. 2 the type of bug in the program, Col. 3 gives the size of the program in lines of code, and Col. 4 gives the number of test cases.

- **Search space:** How should the representation weight program statements to best define the search space? (Section 6.4.5)

These questions all directly evaluate one or more critical (and in many cases novel) components of the GenProg algorithm; their answers help us improve its performance and understand more about why and how it works. Section 6.4.1 describes the benchmark sets used in these experiments and reiterates the default parameter and algorithmic components used as a common baseline in the subsequent experiments. This configuration follows the experiments performed and evaluated in the previous chapters; we repeat it here for clarity of presentation. The remainder of the section provides results.

6.4.1 Benchmarks and baseline parameters

This subsection describes the benchmark sets and establishes baseline parameters and operators for our experiments.

Table 6.2 shows the first set of ten benchmark programs, taken from our earlier work that introduced the AST/WP representation for program repair [217]. These benchmarks are a subset of those presented in Chapter 4, used to evaluate expressive power. We include these benchmarks because the AST/WP representation does not scale to many of the bugs from the ManyBugs benchmark set — the smaller benchmarks thus allow direct comparisons between PATCH and AST/WP on the dataset on which it was originally proposed and evaluated. For the other experiments, we use the ManyBugs benchmark set from Chapter 5, which we prefer where possible because of its size and diversity.

For the purposes of comparison, we define default parameters and operator choices. These correspond to the algorithm as presented in Chapter 3 and evaluated subsequently. We reiterate them here for the purposes of clarity. The

mutation operators are **replace**, **delete**, and **insert**.[1] The population size is 40. The GP is run for a maximum of 10 generations or 12 wall-clock hours, in the Amazon EC2 cloud computing environment, whichever comes first. The tournament size is 2. The mutation rate is 1 mutation per individual per generation. Each individual variant undergoes crossover once per generation, regardless of the crossover operator. When a mutation is applied, one of the mutation operators is selected with equal random probability. The fault localization scheme assigns a weight of 1.0 to statements executed by only the failing test case and 0.1 to statements executed by both the failing negative test case and passing positive test cases (W_{Path}). The fix localization scheme includes the *semantic check* described earlier. $SampleFit$ samples a random 10% of the positive tests for the larger benchmarks. W_{PosT} and W_{NegT} are set to weight the negative test case(s) twice as highly as the positive test cases.

Using this "baseline" parameter set, GenProg repairs 10/10 bugs in Table 6.2 55/105 of the ManyBugs defects, as shown in Section 5.4. Although the relatively small number of trials limits the statistically significant conclusions we can draw, it is still informative to note that the GP success rate varies considerably between the 55 patched bugs. On 21/55 bugs, all 10 default repair trials succeeded; on 24/55, more than 50% but fewer than 100% of the trials succeeded; and on 10/55, less than 50% but more than 0% of the trials succeeded. We observe the general trend (again without drawing strong conclusions), that some of these bugs appear "easier" to repair than others. These distinctions clarify the results presented in several of the subsequent subsections.

The primary metrics in all experiments are success rate (fraction of trials that produce a successful repair) and average number of fitness evaluations to find a repair. Results are averaged over a number of repair trials. Results on the smaller benchmarks in Table 6.2 are computed over 100 trials per tested repair scenario. The number of trials per bug for experiments using the larger benchmarks in ranges from 10 to 20. We limited the number of trials to conserve resources, as we ran these experiments in a commercial cloud environment. We performed sufficient trials per experiment to report statistically significant results, and we report significance in terms of α (probability of an outcome under the null hypothesis) where applicable.

6.4.2 Representation

In this subsection, we directly compare the PATCH and AST/WP representations. We ran GenProg on each benchmark in Table 6.2, testing both representations with and without the WP One-Point crossover operator (for direct comparison) and with and without the semantic check operator. We implement the WP one-point crossover in the PATCH representation by mapping the edits in the patch list to their corresponding statements in what would be the Weighted Path in the AST/WP representation (determined via the execution of the test cases), choosing a point along that path, and crossing over the edits that affect statements before and after that point. We held population size, number of generations, mutation

[1] A parameter set that uses **swap** instead of **replace** leads to comparable results; we thus use include **replace** to maintain consistency with previous experiments.

Program	Fault	LOC	Success Ratio
`gcd`	infinite loop	22	1.07
`uniqw-utx`	segfault	1146	1.01
`look-utx`	segfault	1169	1.00
`look-svr`	infinite loop	1363	1.00
`units-svr`	segfault	1504	3.13
`deroff-utx`	segfault	2236	1.22
`nullhttpd`	buffer exploit	5575	1.95
`indent`	infinite loop	9906	1.70
`flex`	segfault	18775	3.75
`atris`	buffer exploit	21553	0.97
average		**6325**	**1.68**

Table 6.3: The final column reports the ratio of successful repairs found by our GenProg with the PATCH representation enhanced as compared to the AST/WP representation, on the original AST/WP benchmarks. Higher ratios indicate that PATCH outperforms AST/WP; lower ratios indicate the opposite.

Representation	Crossover	Semantic Check	Success Ratio
AST/WP	Yes	No	0.85
	No	Yes	0.94
	Yes	Yes	1.00
PATCH	Yes	No	0.95
	No	Yes	1.01
	Yes	Yes	1.14

Table 6.4: Repair success ratios between AST/WP and PATCH representation, with and without crossover and semantic check. Success ratios are normalized to the best performance of the AST/WP representation (with crossover and the semantic check). Higher ratios indicate that PATCH outperforms AST/WP; lower ratios indicate the opposite.

rate and fault localization strategy constant, changing only the internal representation and genetic operators. We ran 100 random repair trials per benchmark.

Table 6.3 and Table 6.4 show results in terms of success rate ratio.[2] Table 6.3 shows that the new PATCH representation outperforms AST/WP on all benchmarks except `atris`, where success drops slightly, and `look`, where both approaches succeed on all trials. Averaged over these benchmarks, the PATCH representation allows GenProg to find repairs 68% more frequently than AST/WP. This result is consistent with our hypothesis that a patch-based representation enable a more efficient search for solutions than one based on manipulating ASTs directly.

Table 6.4 show results that compare the representations with and without the various tested features. The table normalizes success ratios to a AST/WP representation that includes both crossover and the semantic check, because it offered the best performance for that representation choice. The semantic check strongly influences the success rate of both representations, more so than the inclusion of crossover. Crossover does impact success for both representation

[2] Average number of fitness evaluations to find a repair is equivalent between representations on this benchmark set.

Crossover Operator	Success Rate	Fitness Evaluations to a Repair
No Crossover	54.4%	82.43
Patch Subset	61.1%	163.05
WP One-Point	63.7%	114.12
Patch One-Point	65.2%	118.20

Table 6.5: Success rates and effort to repair for different crossover operators using the PATCH representation. Higher success rates are better; lower number of fitness evaluations to find a repair are better.

choices, however; we investigate it in more detail in the next subsection.

Overall, PATCH outperforms AST/WP. We conclude that the PATCH representation, in addition to its scalability advantages for larger programs, provides a significant advantage over the AST/WP representation for the purposes of source-level evolutionary program repair. This supports our initial argument in favor of this representation choice, first presented in Section 3.4.1.

6.4.3 Crossover

The results in the previous subsection indicate that the PATCH representation outperforms AST/WP, even on the smaller benchmarks to which the latter is restricted. The results indicated that the inclusion of crossover impacts repair success, although the time to repair did not vary significantly on the smaller benchmarks.

We investigate this effect in more detail by comparing different crossover operators for the PATCH representation on the larger and more indicative ManyBugs benchmark suite. We compare **Patch Subset**, **Patch One-Point**, **WP One-Point** (simulated in the patch representation as described in Section 6.4.2), and **No Crossover** repair scenarios. To isolate the influence of crossover, we restricted attention to programs with runs (using default parameters) that produced minimized repairs consisting of multiple mutations (suggesting that crossover might be important). We then reran GenProg on each such program, testing the different crossover operators.

Table 6.5 reports results in terms of GP success rate (higher is better) and repair effort (fitness evaluations; lower is better). **Patch One-Point** crossover finds repairs more often than all other options ($\alpha < 0.05$), and the difference in repair time between it and **WP One-Point** is not statistically significant. By contrast, the **Patch Subset** operator leads to significantly longer repair times ($\alpha < 0.05$). Interestingly, leaving out crossover reduces repair time ($\alpha < 0.05$), but also significantly reduces the GP success rate ($\alpha < 0.01$). A manual inspection of the results suggests that the bugs on which repair takes longer are also less likely to be repaired without crossover. Crossover acts as a macro-mutation operator and provides the explicit evolution mechanism in a GP; it appears that the more difficult search problems (as defined by both success rate and time to repair) benefit the most from these mechanisms.

Program	Operators per Fitness Eval. Insert	Delete	Replace	Operators per Repair Insert	Delete	Replace
`gcd`	0.53	0.08	0.20	3.75	0.00	0.00
`uniq`	0.29	0.53	0.50	0.00	1.67	0.66
`look-u`	0.44	0.95	0.96	0.22	1.78	0.44
`look-s`	0.49	0.71	1.44	0.60	1.20	1.20
`units`	0.16	0.16	0.28	1.67	1.00	1.34
`deroff`	1.10	1.34	2.10	0.00	4.67	0.00
`nullhttpd`	1.27	1.24	2.88	0.00	4.50	0.00
`indent`	1.49	1.52	3.22	2.00	8.17	2.66
`flex`	0.40	0.41	0.80	0.00	2.80	0.00
`atris`	0.34	0.33	0.40	0.00	2.00	0.00
average	0.65	0.73	1.28	0.81	2.78	0.63

Table 6.6: GP Operators: The "Operators per Fitness Eval" columns (Insertions, Deletions, and Replacements) show the total average number of genetic changes per individual between fitness evaluations for each evaluated program variant, in units of genetic operations. The "Operators per Repair" columns report the average number of times each type of evolutionary operation was used in the evolution of the first primary repair discovered in each successful program run. The "average" row should be taken to suggest a trend, acknowledging that the data is too noisy to be significant.

These results suggest that **Patch One-Point** crossover is preferable because it affords the best compromise between success rate and repair effort. It performs comparably to **WP One-Point** crossover in terms of repair effort (and is about 30% faster than the **Patch Subset** operator on this dataset), but significantly outperforms the other operators in terms of success rate (by 2–10%).

6.4.4 Operators

This subsection quantifies the contribution of the different mutation operators in successful repairs and empirically evaluates the effect of an unequal operator selection probability function.

By default, GenProg selects between the three mutation types with equal random probability. However, the operations do not appear equally often in the repairs that result from successful runs. Table 6.6 reports data on the number of genetic operations involved per fitness evaluation and per successful repair for the smaller benchmarks in Table 6.2. The average number of genetic operations per fitness evaluation is 2.66, and the average number of operations to produce a successful repair is 4.22. Summing over all individuals in the population (40) and considering that a repair is discovered on average within 3.6 generations on this dataset, the search on average requires 667 genetic operations to discover the initial repair on these benchmarks. The contribution of the individual operations is very noisy, and it is difficult to draw significant conclusions based on this data. However, the **delete** operator appears to be the most effective, acknowledging that its average is skewed by one example: `indent`.

Table 6.7 reports data, both for initial and for minimized repairs, for the ManyBugs dataset. The operator distribution for initial repairs on this benchmark set is closer to uniform (**insert** : **delete** : **replace** :: 1 : 1.7 : 1.45) than is observed

Program	Initial Repairs Insert	Delete	Replace	Final Repairs Insert	Delete	Replace
`fbc`	1.00	5.00	3.00	0.00	1.00	1.00
`gmp`	2.50	1.00	2.00	0.00	0.50	1.00
`gzip`	0.63	0.88	0.38	0.38	0.38	0.25
`libtiff`	0.54	0.97	0.76	0.25	0.39	0.47
`lighttpd`	0.61	1.22	1.14	0.04	0.31	0.65
`php`	0.26	0.46	0.48	0.18	0.39	0.45
`python`	0.33	0.70	0.17	0.00	0.80	0.20
`wireshark`	0.60	0.70	0.80	0.22	0.33	0.56
average	0.51	0.85	0.74	0.13	0.51	0.57

Table 6.7: GP operator frequency per repair on the baseline parameter set for the ManyBugs benchmarks. "Initial Repairs" reports the frequency of insertions, deletions, and replacements in initial repairs; "Final Repairs" reports the same information for minimized repairs.

on the smaller benchmark set, though again, the data is quite noisy. For final repairs, however, the distribution appears much more strongly skewed in favor of deletions and replacements ($1 : 3.84 : 4.3$).

These results suggest that the different mutation types are not equally important to the repair search; their use in final repairs strongly favors deletions and replacements, despite the fact that the default parameter set encourages a uniform distribution. To test this hypothesis, we modified the operator selection probabilities to match the observed distribution (using 10-fold cross-validation [222] to mitigate the threat of overfitting). We then ran new repair trials on a subset of the ManyBugs benchmarks, measuring time to repair and GP success rate with the modified selection probabilities.

Results are shown in Figure 6.3 (success rate) and Figure 6.4 (repair time). Both figures show that results vary significantly by the initial difficulty of the search problem, as measured by success rate using the default parameter set; we bin the results accordingly for clarity of presentation. This setup did not lead to repairs to previously unrepaired bugs. For bugs that GenProg repairs "easily" with the default parameter set, modifying the operator selection function does not significantly impace performance. For these bugs, GP success rate does not change between the operator selection schemes, and the repair time decrease associated with the non-uniform selection scheme is not statistically significant. Similar trends are seen for the "medium" difficulty bugs, and thus over the entire dataset, since these two classes dominate the "hard" bugs. However, tuning mutation operator selection does significantly improve GenProg's performance on the more difficult bugs, defined as those on which the technique is least sucessful using the default parameter set. The modified scheme led to increased success rate and decreased repair time ($\alpha < 0.05$ for both), by 8.9% and 40% on this data, respectively.

On balance, the search benefits from tuning the operator selection function. We conclude that a non-uniform mutation operator selection scheme that favors deletions and replacements over insertions is beneficial. This scheme

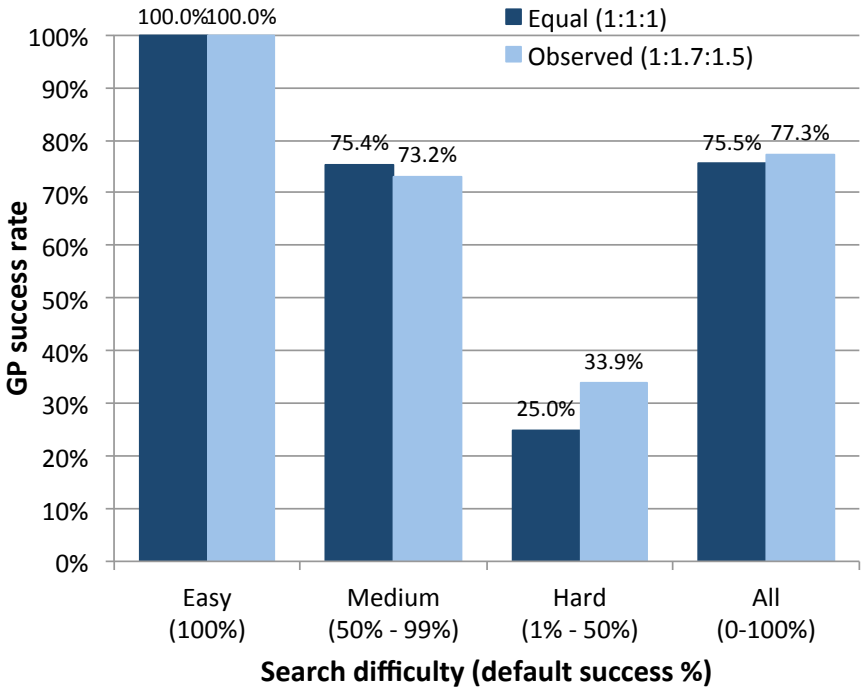

Figure 6.3: GP success rate for two mutation operator selection functions, binned by initial default success rate. Higher is better. The "Equal" selection function chooses between possible mutation operators with equal random probability. The "Observed" selection function is weighted to match the observed distribution of operators in automatic repairs. The "Observed" mutation function succeeds more often on bugs with a low initial success rate ($\alpha < 0.05$).

does not significantly impact performance of the bugs on which GenProg succeeds more often by default. However, it does significantly improve GenProg's performance for the "difficult" searches, which is where the algorithm needs the most help.

6.4.5 Search Space

This subsection studies the weighting scheme used to direct mutation operators to particular areas of a program under repair. It is intuitively appealing that statements executed exclusively by the negative test cases are the most likely source of error, and therefore should be the target of genetic modifications. This insight underlies our approach to fault localization as initially presented in Chapter 3, and used in the experiments so far. The default parameter set weights such statements ten times more highly than those executed by both negative and positive test cases.

Table 6.8 classifies the statements modified in actual repairs according to whether they are executed exclusively by the negative test case or by both the positive and the negative test cases.[3] The data are extremely noisy. However, we observe that the vast majority of repairs do not follow the 10:1 ratio of the default weighting scheme. Over all repairs,

[3] We report data for initial (unminimized) repairs; the distribution for minimized repairs is similar.

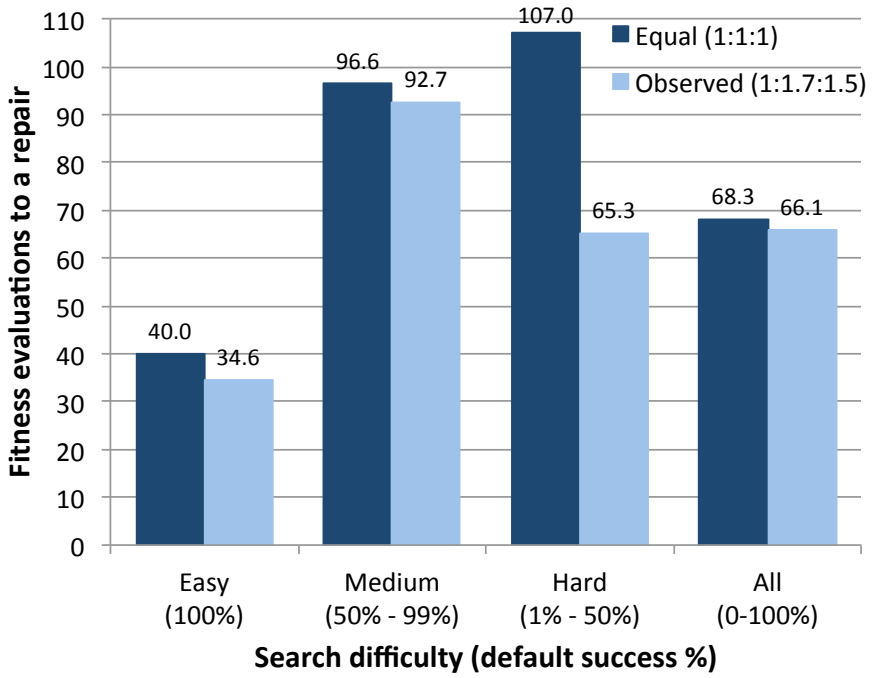

Figure 6.4: Fitness evaluations to a repair for two mutation operator selection functions, binned by success rate with default parameters. Lower is better. The "Equal" selection function chooses between possible mutation operators with equal random probability. The "Observed" selection function is weighted to match the observed distribution of operators in automatic repairs. The bugs with the lowest initial success rate converge more quickly with the "Observed" mutation probabilities ($\alpha < 0.05$)

the ratio of statements executed solely by the negative test cases to those executed by both negative and positive test cases averages to $1 : 1.85$.

These results suggest that the 10:1 path weighting scheme may not be optimal. To test this hypothesis, we tried two additional weighting schemes: a *Realistic* weighting scheme approximating the observed average, and an *Equal* weighting scheme, in which all statements along the negative path receive weight 1.0 (to mitigate against overfitting, as the data are extremely noisy, and to establish a baseline).

Figure 6.5 and Figure 6.6 show results in terms of success rate and number of fitness evaluations needed to find a repair. As with previous experiments, results are more striking for more difficult repair scenarios. Both new weighting schemes led to repairs of a bug on which GenProg failed using the default weighting scheme. With the exception of searches on which the default weighting scheme achieved 100% success, the alternative weighting schemes significantly ($\alpha < 0.05$ for both metrics) outperform the default scheme: success rates increase, and repair times decrease. The differences on the 100% bugs, while statistically significant, are small in terms of both success rates and wall-clock time. On the other hand, the new weighting schemes require only 25% and 50% of the time taken by the default weighting scheme on the more difficult repairs.

In practice, automatically-generated repairs do not modify the "exclusively negative" statements ten times more

Program	Number of repairs	Negative Only	Negative or Positive
`fbc`	1	50%	50%
`gmp`	2	14%	86%
`gzip`	8	36%	64%
`libtiff`	136	72%	28%
`lighttpd`	34	99%	1%
`php`	181	52%	48%
`python`	6	71%	29%
`wireshark`	9	100%	0%

Table 6.8: Location of successful repairs. "Num Repairs" shows the number of unique repairs. "Negative Only" reports the average percentage of the statements modified by a repair that are executed only by the negative test case. "Negative or Positive" is similar, but reports percentages for statements executed by both positive and negative test cases.

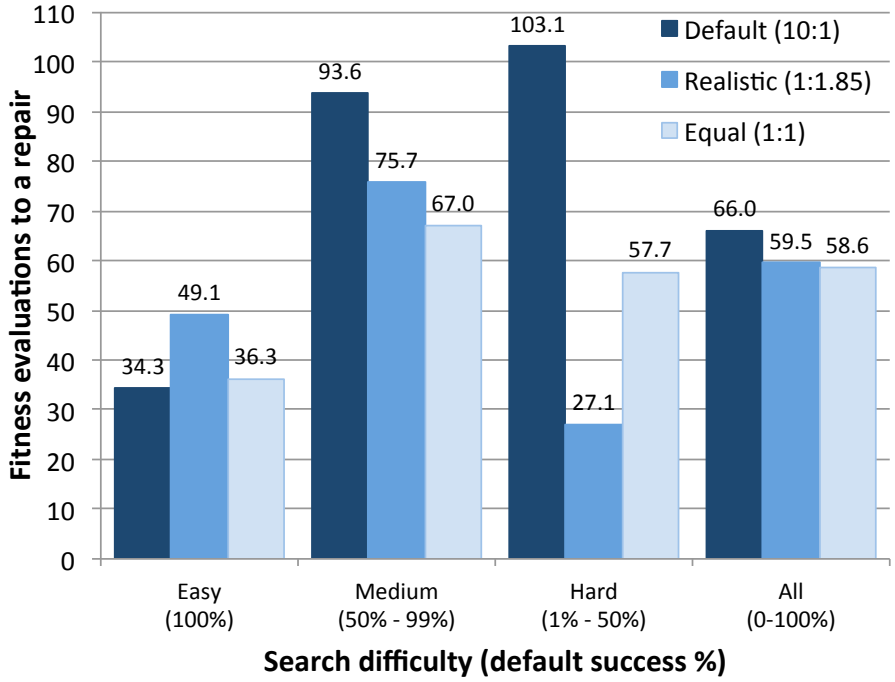

Figure 6.5: Repair effort (number of fitness evaluations to first repair) for the "Default", "Realistic", and "Equal" path weighting schemes, binned by success on the default parameter set. Lower is better. Both the Equal and Realistic weighting take fewer fitness evaluations ($\alpha < 0.01$) to repair the more difficult bugs.

often than they do the "negative and positive" statements. On our dataset, a version of GenProg that uses either of the alternative weighting schemes provides performance comparable to the default configuration (using the 10:1) ratios on those scenarios where it succeeds easily. The alternative weighting outperforms the default, in some cases considerably, on more "difficult" bugs, to the point of repairing an additional bug from the dataset.

We provide final remarks on these experiments in the next section, which summarizes the results of the chapter.

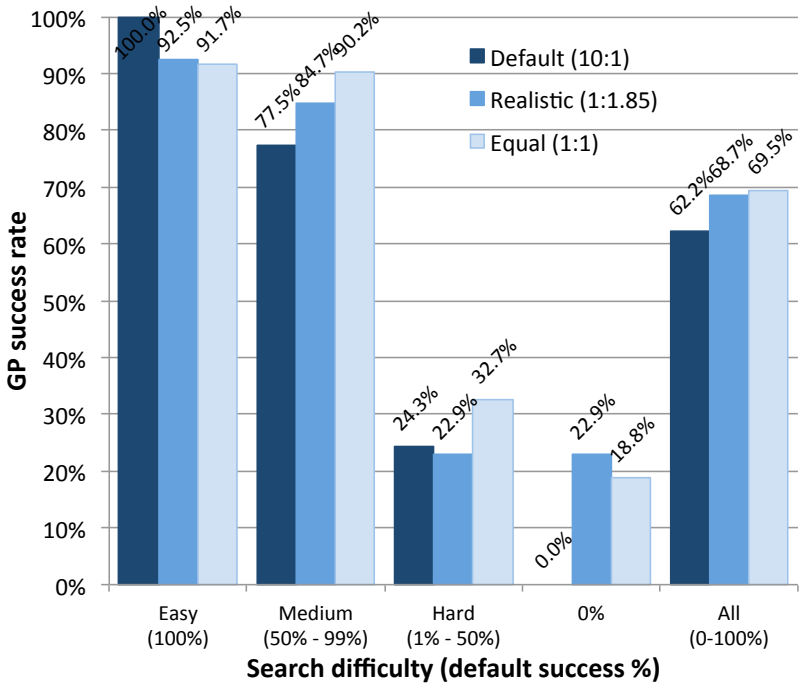

Figure 6.6: Repair success for the "Default", "Realistic", and "Equal" path weighting schemes, binned by success on the default parameter set. Higher is better for success rate. Both the Equal and Realistic weighting schemes outperform Default in terms of overall success rate ($\alpha < 0.05$).

6.5 Summary and discussion

In this chapter, we analyzed the repair results in Chapter 4 and Chapter 5, focusing specifically on performance and scalability concerns. We observe that intermediate fitness evaluations dominate execution time, and substantiated the fact that GenProg, in general, benefits from a test suite that can be run quickly. This helps motivate our use of $SampleFit$ in our larger benchmarks. On both sets of benchmarks in the previous chapters, GenProg repair time appears to scale sub-linearly with a defect's fault space size. Additionally, fix space size also appears to influence repair time, likely because a larger fix space corresponds to more options for a given repair. Finally, we found no observable relationship between external measures of bug severity and likelihood of repair or repair time, suggesting that GenProg is broadly applicable across a wide range of bugs, and not just those that are "low-severity." This observation suggests that there may be a weak or no correlation between bug severity and fix complexity, a notion that is corroborated by the security vulnerabilities we have examined, fixed, and explained in this dissertation. Often the most severe of security vulnerabilities can be fixed by adding a bounds check.

These results motivated a large-scale study evaluating different representation, operator, and search space choices in the context of evolutionary program repair. We conclude with four concrete observations about the design decisions that underlie the algorithm proposed and evaluated in this dissertation.

First, in addition to the scalability benefits afforded by storing individual variants as small lists of edits instead of entire program source trees, the PATCH representation is more effective than AST/WP in terms of repair success. The semantic check contributes importantly to repair success regardless of representation, more so than inclusion of crossover.

Second, a one-point crossover operator over the patch representation (**Patch One-Point**) offers the best tradeoff between repair success and time. It found repairs 28% faster than the **Patch Subset** crossover operator with a 4% improvement in success rate. Omitting crossover also decreases repair time, but it does so at the expense of success rate. This suggests that the scenarios that take longer to repair are also those for which the crossover operator is the most important.

Third, **delete** is the most useful mutation operator of those we tested, followed by **replace**, followed by **insert**. A biased mutation selection technique improves both success rate and repair time, particularly for more challenging repairs. On these examples, a biased operator selection function improved success rate from 25.0% to 33.9%, and decreases repair time by 40%.

Finally, the assumption that statements executed exclusively by negative test cases should be weighted much more heavily than those executed by both the positive and negative test cases is flawed. The actual distribution of modifications in final repairs, by and large, does not follow this assumption, despite the fact that the default weighting scheme is expected to bias the repairs in that direction. Changing the distribution improves both success rates and repair time, particularly on the more challenging bugs; on such examples, time to repair improves by up to 70%. Doing so also allows GenProg to repair bugs on which the default setup fails.

We have attempted to mitigate several threats to the validity of the results and analyses in this chapter. In the pursuit of a controlled study, we made heuristic choices for a number of parameters, such as population size, tournament size, and mutation rate. We attempted to record these choices clearly. To guard against the risks of inappropriate or unindicative statistical results, we used nonparametric tests on smaller datasets to mitigate the risk of data that are not normally distributed. In the analyses in the first part of the chapter, there were instances in which we were unable to identify statistical relationships, for example between repair time and program size; this does not mean that these relationships do not exist, only that we were unable to uncover them. This threat is mitigated by the consistency and significance of the relationships that we did identify across the different datasets. While our computational setup limits the number of trials we can run, we quantify α wherever applicable to increase confidence that the results are significant. We use cross-validation to mitigate the threat of overfitting where applicable and a uniform baseline for the search space experiments, when the distribution is too noisy to be significant.

Taken together, these results suggest first, that many of the algorithmic design choices we made in Chapter 3 are supported by the empirical evidence. However, it is profitable to examine such assumptions, because operator choice, program representation, and probability distributions do individually matter, especially for difficult bugs. Additionally,

not all of our intuitions played out in practice: Our fault space characterization, notably, may be suboptimal according to these results.

Although the features that contribute to bug repair difficulty warrant further study, the results in this chapter suggest strongly that there are clear distinctions, and the more difficult examples are the most sensitive to such choices. In the end, these apparently disjoint features combine synergistically. When we modify our default parameter set with the recommendations outlined above and rerun the repair scenarios from Table 5.1, GenProg automatically repairs 5 additional bugs (60 vs. 55), and repair time decreases by 17–43% for the more "difficult" search scenarios.

Chapter 7

Conclusions

> "If finding bugs is technically demanding and yet economically vital, how much more difficult yet valuable would it be to automatically fix bugs?"
>
> – *Mark Harman [223]*

Bugs remain a dominant challenge in software engineering research and practice. Our overall goal in the research presented in this dissertation is to reduce the costs associated with defect repair in software maintenance. To that end, we proposed GenProg, a technique designed to automatically fix bugs in legacy systems, and presented empirical evidence to substantiate our claims about its effectiveness along several dimensions. We summarize the dissertation and its contributions in Section 7.1, and provide final remarks in Section 7.2.

7.1 Summary

We began by surveying the state of industrial practice to identify the properties necessary in a viable technique for automatic program repair:

- **Scalability.** A technique for automated repair should cheaply scale to the large, realistic software systems that exist in practice.
- **Quality repairs.** A technique for automated repair should produce quality bug fixes that address the root of the problem and that do not introduce new crashes or vulnerabilities.
- **Expressive power.** A technique for automated repair should be able to fix a variety and a large proportion of the defects that appear in practice.

The primary thesis that guided the work presented in this dissertation was thus:

> **Thesis:** stochastic search, guided by existing test cases, can provide a scalable, expressive, and human competitive approach for the automated repair of many types of defects in many types of real-world programs.

We believe this thesis to be possible, as evidenced by the algorithm we constructed and the results we obtained evaluating it. GenProg embodies a combination of genetic programming, execution-based testing, and lightweight program analyses designed to accomplish viable automated program repair. It is based on three broad insights: (1) the space of software patches is amenable to exploration via stochastic search, (2) test cases provide useful insight into program behavior and (3) existing program behavior contains the seeds of many repairs.

The properties that served as our design goals, coupled with our insights about the nature of software and bug fixes, informed our decisions in designing the algorithm. GenProg's application of genetic programming is very different from other work that uses GP, and its internals are novel in several ways:

- A compact, scalable, and expressive representation of candidate solutions.
- Efficient and domain-specific crossover, mutation, and selection operators for manipulating and traversing the space of candidate solutions.
- A formalization of the program repair search space, including the novel application of fault localization to this domain and the introduction of the idea of fix localization. This conception of the search space is novel on its own, and also enables the algorithm's scalability by intelligently constraining the search.
- A technique to efficiently yet correctly guide the GP search for a repair by sampling test suites.
- A novel application of software engineering debugging and tree-based differencing techniques to overcome genetic programming's classic code bloat problem as it manifests in this application.

The properties we sought in an automatic repair solution informed four overarching hypotheses that guided our evaluation of GenProg:

1. Without defect- or program-specific information, GenProg can repair at least 5 different defect types and can repair defects in at least least 10 different program types.
2. GenProg can repair at least 50% of defects that human developers fix in practice.
3. GenProg can repair bugs in programs of up to several million lines of code, and associated with up to several thousand test cases, at a time and economic cost that is human competitive.
4. GenProg-produced patches maintain existing program functionality as measured by existing test cases and held-out indicative workloads; they do not introduce new vulnerabilities as measured by black-box fuzz testing

techniques; and they address the underlying cause of a vulnerability, and not just its symptom, as measured against exploit variants generated by fuzzing techniques.

We tested these hypotheses through a series of empirical evaluations. These evaluations required the design and construction of new benchmark sets, notably the ManyBugs set presented in Section 5.3, as well as new experimental frameworks for evaluating repair quality (Section 4.4) and human-comparable time and monetary costs (Section 5.1). Our results:

- Demonstrated GenProg's ability to repair bugs covering 8 different types of defects in 16 programs (Section 4.2).
- Qualitatively (Section 4.3 and Section 5.6) and quantitatively (Section 4.4) substantiated our claims that the repairs GenProg proposes are of acceptable quality.
- Evaluated GenProg's ability to repair 52% of a large set of defects that are indicative of those that human developers address in practice (Section 5.4).
- Demonstrated that the costs to repair those defects are human competitive in both temporal and economic terms (Section 5.4.2); GenProg repaired the bugs in that set for $7.32, including the costs of the failed runs.

We next analyzed GenProg's performance (Chapter 6). We indicated that the dominant factor in its runtime is executing the input test suite (Section 6.1.1), and the dominant factor in its scaling behavior is the size of the search space as parameterized by its fault and fix space (Section 6.1.2). These results are encouraging because they suggest first that GenProg scales sublinearly with program size, and second, there are clear and tractable avenues for improvement of the technique: improving fitness function efficiency, and increasing the precision and accuracy of both fault and fix localization.

Finally, we presented a large-scale study of several of the key representation and operator design choices in the algorithm (Section 6.3). This study resulted in four recommendations about how to effectively run a technique like GenProg, and importantly, overturned an assumption about how the search space should best be characterized. This study does not close the book, and there remain a number of open research questions about the components of our algorithm and the parameters with which it is run. However, the results substantiated our belief that this search problem is atypical; atypical problems warrant special empirical consideration; and the results from empirical studies of this kind can overturn long-standing assumptions. The study also demonstrated that there are clear distinctions in the difficulty of the bugs that humans fix in practice and that comprise our dataset: some constitute "more difficult" search problems than others, and our underlying algorithmic design decisions matter the most for these more difficult bugs. Why or in what way those bugs are different from the others is another open question, whose answer will likely provide useful insight about next steps for research in automatic program repair.

Venue	Publication	Notes
ICSE '09	*Automatically finding patches using genetic programming* [217]	Distinguished Paper Manfred Paul Award Gold, Humies 2009 [37]
GECCO '09	*A genetic programming approach to automated software repair* [221]	Best paper Gold, Humies 2009 [37]
CACM '10	*Automatic program repair with evolutionary computation* [224]	Invited
GECCO '10	*Designing better fitness functions for automated program repair* [201]	
TSE '12	*GenProg: A Generic Method for Automated Software Repair* [215]	Distinguished article
ICSE '12	*A Systematic Study of Automated Program Repair: Fixing 55 out of 105 bugs for $8 Each* [219]	Bronze, Humies 2012 [37]
GECCO '12	*Representations and Operators for Improving Evolutionary Software Repair* [225]	Bronze, Humies 2012 [37]

Table 7.1: The major publications that support this dissertation. The ACM SIGEVO Humie awards recognize human competitive results of an application of evolutionary computation, and are awarded to a set of papers in a year. The IFIP TC2 Manfred Paul Award for "excellence in software: theory and practice" is awarded annually to one paper selected from one of several conferences.

7.2 Discussion, impact, and final remarks

Our personal communication with industry practitioners suggest that there is real-world potential in the techniques we proposed and evaluated here. The time and economic cost of bugs in software are so great at present that we see genuine room for growth and adoption, if not of this technique specifically, than of its successors generally.

In the short term, although GenProg currently requires test cases and, in most reasonable applications, developer validation of candidate repairs, our results suggest that it can quite clearly reduce the immediate cost of fixing bugs in real systems. Even if the human developer remains in the loop, or if patches are not fully trusted in the absence of human inspection, these results still have implications for the future of automated program repair and software engineering more generally. For example, part of the high cost of developer turnover may be mitigated by using the time saved by this technique to write additional tests, which remain even after developers leave, to guide future repairs. GenProg could also be used to generate fast, cheap repairs that serve as temporary bandages and provide time and direction for developers to find longer-term fixes.

This dissertation serves as a detailed description of the state of the art for very early automated software repair efforts. We hope that the approach and its associated tools, benchmarks, and experimental plans that we designed to evaluate it will help serve as a mere starting point for research in automatic program repair. We have released the source code and experimental scenarios publicly, to encourage both controlled reproduction of our results as well as, more importantly, empirical experiments on new and better approaches to tackling the software quality problem.

In the long term, we believe that the work in this dissertation is but a first step in a new an exciting research area.

The primary findings presented in this document have been published at major software engineering and evolutionary computation research venues (see Table 7.1). Those publications and the associated results have been honored by several awards, particularly in acknowledgement of the novel human competitive results that GenProg obtains on real bugs in real systems. Over the five or so years during which this work has been conducted, general automated software repair has gone from being entirely unheard of to having its own multi-paper sessions in top tier Software Engineering conferences (e.g., the 2013 International Conference on Software Engineering). The research area shows no sign of slowing down.

Bibliography

[1] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, pages 141–154, 2003.

[2] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39, 2005.

[3] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Technical Report NIST Planning Report 02-3, NIST, May 2002.

[4] Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Workshop on Mining Software Repositories*, May 2007.

[5] Leigh Williamson. IBM Rational software analyzer: Beyond source code. In *Rational Software Developer Conference*, June 2008.

[6] Symantec. Internet security threat report. In *http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf*, September 2006.

[7] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Automated Software Engineering*, pages 34–43, 2007.

[8] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. How do fixes become bugs? In *Foundations of Software Engineering*, pages 26–36, 2011.

[9] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[10] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic Internet services. In *International Conference on Dependable Systems and Networks*, pages 595–604, 2002.

[11] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. BugFix: A learning-based tool to assist developers in fixing bugs. In *International Conference on Program Comprehension*, 2009.

[12] Stelios Sidiroglou, Giannis Giovanidis, and Angelos D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *Information Security*, pages 1–15, 2005.

[13] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, 2009.

[14] Joel Jones. Abstract Syntax Tree Implementation Idioms. Technical report, 2003.

[15] S. Gustafson, A. Ekart, E. Burke, and G. Kendall. Problem difficulty and code growth in genetic programming. *Genetic Programming and Evolvable Machines*, pages 271–290, 2004.

[16] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[18] Certified Tester Foundation Level Syllabus. Technical report, Board of International Software Testing Qualifications, 2011.

[19] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on dependable and secure computing*, 1(1):11–33, January 2004.

[20] Robert Richardson. FBI/CSI computer crime and security survey. Technical report, http://www.gocsi.com/forms/csi_survey.jhtml, 2008.

[21] Melinda-Carol Ballou. Improving software quality to drive business agility. White paper, International Data Corporation, June 2008.

[22] Christian Robottom Reis and Renata Pontin de Mattos Fortes. An overview of the software engineering process and tools in the Mozilla project. In *Open Source Software Development Workshop*, pages 155–175, 2002.

[23] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2nd edition, July 2004.

[24] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *Operating Systems Design and Implementation*, pages 303–316, 2004.

[25] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis*, 2006.

[26] Sumit Gulwani. Dimensions in program synthesis. In *Principles and practice of declarative programming*, pages 13–24, 2010.

[27] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages*, pages 1–3, 2002.

[28] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *USENIX Security Symposium*, 2006.

[29] Dorothy E. Denning. An intrusion-detection model. *IEEE Trans. Software Eng.*, 13(2):222, February 1987.

[30] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to the conference on Object-oriented programming systems, languages, and applications*, pages 132–136, 2004.

[31] Lars Albertsson and Peter S. Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workload. In *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page 191, 2000.

[32] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: Explaining program failures via postmortem static analysis. In *Foundations of Software Engineering*, 2004.

[33] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[34] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity*, pages 500–517, 2001.

[35] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with Jolt. In *European Conference on Object Oriented Programming*, 2011.

[36] John Henry Holland. Genetic algorithms. *Scientific American*, pages 66–72, July 1992.

[37] John R. Koza. Awards for human-competitive results produced by genetic and evolutionary computation. http://www.genetic-programming.org/hc2009/cfe2009.html, 2009.

[38] Amy K. Hoover, Paul A. Szerlip, and Kenneth O. Stanley. Interactively evolving harmonies through functional scaffolding. In *Genetic and Evolutionary Computation Conference*, pages 387–394. ACM, 2011.

[39] Richard A. J. Woolley, Julian Stirling, Adrian Radocea, Natalio Krasnogor, and Philip Moriarty. Automated probe microscopy via evolutionary optimization at the atomic scale. *Applied Physics Letters*, 98:253104–253104–3, June 2011.

[40] Achiya Elyasaf, Ami Hauptman, and Moshe Sipper. Evolutionary design of freecell solvers. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(4):270–281, 2012.

[41] Stephanie Forrest. Genetic algorithms: Principles of natural selection applied to computation. *Science*, 261:872–878, Aug 1993.

[42] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[43] 36 human-competitive results produced by genetic programming. http://www.genetic-programming.com/humancompetitive.html, Downloaded Aug. 17, 2008.

[44] Andrea Arcuri, David Robert White, John Clark, and Xin Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *International Conference on Simulated Evolution And Learning*, pages 61–70, 2008.

[45] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Programming Language Design and Implementation*, pages 234–245, 2002.

[46] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Programming Language Design and Implementation*, pages 213–223, 2005.

[47] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *Transactions on Software Engineering*, 38(2):278–292, 2012.

[48] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, 2001.

[49] Yue Jia and Mark Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic & Industrial Conference*, pages 94–98, 2008.

[50] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis*, pages 177–187, 2012.

[51] Jean-Claude Laprie, Algirdas Avizienis, and Hermann Kopetz, editors. *Dependability: Basic Concepts and Terminology*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992.

[52] John C. Knight and Paul E. Ammann. An experimental evaluation of simple methods for seeding program errors. In *International Conference on Software Engineering*, 1985.

[53] Robert C. Seacord and Allen D. Householder. A Structured Approach to Classifying Security Vulnerabilities. Technical report, U.S. Department of Defense, January 2005.

[54] Matt Bishop and David Bailey. A Critical Analysis of Vulnerability Taxonomies. Technical report, 1996.

[55] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Principles of Programming Languages*, pages 372–382, 2006.

[56] M. Sutton. How prevalent are SQL injection vulnerabilities? Technical report, http://portal.spidynamics.com/blogs/msutton/archive/2006/09/26/How-Prevalent-Are-SQL-Injection-Vulnerabilities_3F00_.aspx, September 2006.

[57] K. J. Higgins. Cross-site scripting: attackers' new favorite flaw. Technical report, http://www.darkreading.com/document.asp?doc_id=103774&WT.svl=news1_1, September 2006.

[58] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb. Security Analysis and Enhancements of Computer Operating Systems. Technical report, U.S. Department of Commerce, Washington, D. C. 20234, USA, April 1976.

[59] Taimur Aslam. A taxonomy of security faults in the UNIX operating system. Master's thesis, Purdue University, 1995.

[60] Richard Bisbey II and Dennis Hollingworth. Protection analysis: Final report. Technical report, USC/ISI, May 1978.

[61] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[62] Misha Zitser, D. E. Shaw Group, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Foundations of Software Engineering*, pages 97–106. ACM Press, 2004.

[63] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[64] Rebecca Isaacs and Yuanyuan Zhou, editors. *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*. USENIX Association, 2008.

[65] Xiang Song, Haibo Chen, and Binyu Zang. Why software hangs and what can be done with it. In *Dependable Systems and Networks*, pages 311–316. IEEE, 2010.

[66] BBC News. Microsoft Zune affected by 'bug'. In *http://news.bbc.co.uk/2/hi/technology/7806683.stm*, December 2008.

[67] Mary Jean Harrold. Testing: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 61–72. ACM, 2000.

[68] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[69] Stephen R. Schach. Testing: Principles and practice. *ACM Comput. Surv.*, 28(1):277–279, 1996.

[70] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411, 2005.

[71] Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. Regression testing in an industrial environment. *Commun. ACM*, 41(5):81–86, 1998.

[72] Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors. *International Conference on Software Testing, Verification and Validation*. IEEE, 2012.

[73] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[74] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[75] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Principles of Programming Languages*, pages 174–186, 1997.

[76] Patrice Godefroid. Compositional dynamic test generation. In *Principles of programming languages*, pages 47–54, 2007.

[77] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[78] Mary Lou Soffa, Aditya P. Mathur, and Neelam Gupta. Generating test data for branch coverage. In *Automated Software Engineering*, page 219, 2000.

[79] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation*, pages 209–224, 2008.

[80] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *International Symposium on Software Testing and Analysis*, pages 151–162, 2007.

[81] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*. Internet Society, 2008.

[82] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, October 2007.

[83] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Foundations of Software Engineering*, pages 263–272, 2005.

[84] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, pages 2–23, 2005.

[85] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Computer and communications security*, pages 322–335, 2006.

[86] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[87] Raúl A. Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. In *Automated Software Engineering*, pages 218–227, 2008.

[88] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *International Symposium on Software Testing and Analysis*, pages 364–374, 2011.

[89] J. Penix. Large-scale test automation in the cloud (invited industrial talk). In *International Conference on Software Engineering*, pages 1122–1122, 2012.

[90] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *USENIX Security Symposium*, pages 67–82, 2009.

[91] Gregg Rothermel, Roland J. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[92] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering Methodology*, 2(3):270–285, 1993.

[93] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *International Conference on Software Engineering*, pages 308–318, 2003.

[94] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *International Conference on Software Engineering*, pages 491–500, 2004.

[95] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Notices*, 39(10):432–448, 2004.

[96] Mark Harman. The current state and future of search based software engineering. In *International Conference on Software Engineering*, pages 342–357, 2007.

[97] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference*, pages 1925–1932, 2006.

[98] Christoph C. Michael, Gary McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[99] Ahilton Barreto, Marcio de O. Barros, and Claudia M.L. Werner. Staffing a software project: a constraint satisfaction and optimization-based approach. *Computers and Operations Research*, 35(10):3073–3089, 2008.

[100] Enrique Alba and Francisco Chicano. Finding safety errors with ACO. In *Genetic and Evolutionary Computation Conference*, pages 1066–1073, 2007.

[101] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Genetic and Evolutionary Computation Conference*, pages 1909–1916, 2006.

[102] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[103] C. A. R. Hoare. An axiomatic basis for computer programming. *Communcations of the ACM*, 12(10):576–580, 1969.

[104] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[105] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, Jun 1992.

[106] Adam Chipala, editor. *The 4th Coq workshop*, 2012.

[107] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[108] Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. In *International Joint Conference on Artificial Intelligence*, pages 241–252, 1969.

[109] Robert L. Constable. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc, Upper Saddle River, NJ, USA, 1986.

[110] D. R. Smith. Kids: A semiautomatic program development system. *IEEE Trans. Softw. Eng.*, 16(9):1024–1043, 1990.

[111] Thomas Emerson and Mark H. Burstein. Development of a constraint-based airlift scheduler by program synthesis from formal specifications. In *Automatic Software Engineering*, page 267, 1999.

[112] Armando Solar-Lezama. The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems*, pages 4–13, 2009.

[113] Vladimir Jojic, Sumit Gulwani, and Nebojsa Jojic. Probabilistic inference of programs from input/output examples. Technical report, MSR-TR-2006-103, Microsoft Research, 2006.

[114] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Programming Language Design and Implementation*, pages 62–73, 2011.

[115] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *International Conference on Dependable Systems and Networks*, pages 52–61, 2008.

[116] Microsoft Corporation. Dr. watson overview, 2002.

[117] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[118] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[119] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[120] Mark Gabel and Zhendong Su. Testing mined specifications. In *Foundations of Software Engineering*, pages 1–11. ACM, 2012.

[121] Claire Le Goues and Westley Weimer. Measuring code quality to improve specification mining. *IEEE Transactions on Software Engineering*, 38(1):175–190, 2012.

[122] Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors. *Recent Advances in Intrusion Detection*, volume 5230 of *Lecture Notes in Computer Science*. Springer, 2008.

[123] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Computer and Communications Security*, pages 251–261. ACM Press, 2003.

[124] Elvis Tombini, Hervé Debar, Ludovic Mé, and Mireille Ducassé. A serial combination of anomaly and misuse idses applied to http traffic. In *20th Annual Computer Security Applications Conference*, 2004.

[125] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advances in Intrusion Detection*, volume 3224 of *Lecture Notes in Computer Science*, pages 203–222, 2004.

[126] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Computer and Communications Security*, pages 272–280, 2003.

[127] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium of Software Testing and Analysis*, pages 218–228, 2002.

[128] Michael E. Locasto, Gabriela F. Cretu, Shlomo Hershkop, and Angelos Stavrou. Post-patch retraining for host-based anomaly detection. Technical Report CUCS-035-07, Columbia University, October 2007.

[129] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, May 2001.

[130] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.

[131] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Static Analysis Symposium*, pages 295–315, 2003.

[132] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. *SIGPLAN Notices*, 38(1):97–105, 2003.

[133] Alex Groce and Daniel Kroening. Making the most of BMC counterexamples. *Electronic Notes in Theoretical Computer Science*, 119(2):67–81, 2005.

[134] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *Foundations of Software Engineering*, pages 73–82, 2004.

[135] S. C. Johnson. Lint, a C program checker. In *Computer Science Technical Report*, pages 78–1273, 1978.

[136] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[137] Stephanie Horstmanshof and Suzanne Ross. http://research.microsoft.com/en-us/news/features/prefast.aspx, October 2004.

[138] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB*. Free Software Foundation, 2002.

[139] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, pages 89–100, 2007.

[140] TD LaToza and Brad A Myers. Developers ask reachability questions. In *International Conference on Software Engineering*, pages 184–194, 2010.

[141] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, June 2005.

[142] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, pages 30–39, 2003.

[143] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[144] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Pacific Rim International Symposium on Dependable Computing*, pages 39–46. IEEE Computer Society, 2006.

[145] Westley Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[146] Denis Kozlov, Jussi Koskinen, Markku Sakkinen, and Jouni Markkula. Assessing maintainability change over multiple software releases. *Journal of Software Maintenance and Evolution*, 20:31–58, January 2008.

[147] Kazuki Nishizono, Shuji Morisaki, Rodrigo Vivanco, and Kenichi Matsumoto. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks — an empirical study with industry practitioners. In *International Conference on Software Maintenance*, pages 473 –481, sept. 2011.

[148] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, 2009.

[149] Andreas Zeller. Automated debugging: Are we close? *IEEE Computer*, 34(11):26–31, 2001.

[150] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In *Fundamental Approaches to Software Engineering*, pages 267–280, 2004.

[151] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. DebugAdvisor: A recommender system for debugging. In *Foundations of Software Engineering*, pages 373–382, 2009.

[152] Alan Turing. *Systems of Logic Based on Ordinals*. PhD thesis, Princeton University, Princeton, NJ, 1939.

[153] Sean Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[154] L.A. Rastrigin. The convergence of the random search method in the extremal control of a many parameter system. *Automation and Remote Control*, 24(10):1337–1342, 1963.

[155] Terence Soule and Jason H. Moore, editors. *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012*. ACM, 2012.

[156] Rafael Martí, Manuel Laguna, and Fred Glover. Principles of scatter search. *European Journal of Operational Research*, 169(2):359–372, 2006.

[157] James Kennedy and Russell Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[158] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671680, 1983.

[159] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. 45:41–51, 1985.

[160] Marco Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italie, 1992.

[161] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, pages 65–74, 2010.

[162] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *Transactions on Evolutionary Computation*, 15(2):166–192, 2011.

[163] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automatic program repair through the evolution of assembly code. In *Automated Software Engineering*, pages 33–36, 2010.

[164] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Genetic and Evolutionary Computation*, pages 1427–1434, 2011.

[165] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(5), 2011.

[166] William B. Langdon and Mark Harman. Evolving a CUDA kernel from an nVidia template. In *Congress on Evolutionary Computation*, pages 1–8, 2010.

[167] David R. White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[168] Michael Orlov and Moshe Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In *Genetic and Evolutionary Computation Conference*, pages 1043–1050, 2009.

[169] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Congress on Evolutionary Computation*, pages 162–168, 2008.

[170] Josh L. Wilkerson, Daniel R. Tauritz, and James M. Bridges. Multi-objective coevolutionary automated software correction. In *Genetic and Evolutionary Computation Conference*, pages 1229–1236, 2012.

[171] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[172] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *International Conference on Software Engineering*, pages 855–858, 2008.

[173] Alexey Smirnov and Tzi-Cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium*, 2005.

[174] Alexey Smirnov, Rachael Lin, and Tzi-Cker Chiueh. PASAN: Automatic patch and signature generation for buffer overflow attacks. In *Systems and Information Security*, pages 165–170, 2006.

[175] Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, and Angelos D. Keromytis. From STEM to SEAD: speculative execution for automated defense. In *USENIX Annual Technical Conference*, pages 1–14, 2007.

[176] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2009.

[177] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[178] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using sat. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2011.

[179] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Automated Software Engineering*, pages 550–554, 2009.

[180] Peng Liu and Charles Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *International Conference on Software Engineering*, pages 299–309, 2012.

[181] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, 2011.

[182] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification Inference for Explicit Information Flow Problems. In *Programming Language Design and Implementation*, pages 75–86, 2009.

[183] Nullhttpd. Bug: http://nvd.nist.gov/nvd.cfm?cvename=CVE-2002-1496. Exploit: http://www.mail-archive.com/bugtraq@securityfocus.com/msg09178.html, 2002.

[184] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, San Francisco, CA, 2013 (To appear).

[185] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *International Conference on Genetic Algorithms*, pages 184–192, 1995.

[186] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[187] Terry Van Belle. *Modularity and the Evolution of Software Evolvability*. PhD thesis, University of New Mexico, Albuquerque, NM, 2004.

[188] Terry Van Belle and David H. Ackley. Code factoring and the evolution of evolvability. In *Genetic and Evolutionary Computation Conference*, pages 1383–1390, 2002.

[189] Robin Harper. Spatial co-evolution: quicker, fitter and less bloated. In *Genetic and Evolutionary Computation Conference*, pages 759–766. ACM, 2012.

[190] Nicholas J. Radcliffe. The algebra of genetic algorithms. *Ann. Math. Artif. Intell.*, 10(4):339–384, 1993.

[191] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[192] Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[193] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *Conference on Human Factors in Computing Systems*, CHI '07, pages 567–576, New York, NY, USA, 2007. ACM.

[194] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: An infrastructure for C program analysis and transformation. In *International Conference on Compiler Construction*, pages 213–228, 2002.

[195] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. December 2011.

[196] Brad L. Miller and David E. Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary Computing*, 4(2):113–131, 1996.

[197] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[198] Jonathan E. Rowe and Nicholas Freitag McPhree. The effects of crossover and mutation operators on variable length linear structures. In *Genetic and Evolutionary Computation Conference*, pages 535–542, 2001.

[199] G. Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *International Conference on Genetic Algorithms*, pages 2–9, 1989.

[200] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, pages 15–26, 2005.

[201] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference*, pages 965–972, 2010.

[202] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, 1999.

[203] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[204] Raihan Al-Ekram, Archana Adma, and Olga Baysal. diffX: an algorithm to detect changes in multi-version XML documents. In *Conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.

[205] Stephen M. Blackburn, Robin Garner, Chris Hoffman, Asad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[206] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and control flow-based test adequacy criteria. In *International Conference on Software Engineering*, pages 191–200, 1994.

[207] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *International Conference on Software Engineering*, pages 121–130, 2011.

[208] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium*, 2006.

[209] Kenneth L. Ingham, Anil Somayaji, John Burge, and Stephanie Forrest. Learning DFA representations of HTTP for protecting web applications. *Computer Networks*, 51(5):1239–1255, 2007.

[210] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[211] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[212] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Programming Language Design and Implementation*, pages 446–455, 2007.

[213] Jonathan D. Pincus. Analysis is necessary, but far from sufficient: Experiences building and deploying successful tools for developers and testers (abstract only). In *ISSTA*, page 1, 2000.

[214] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar T. Devanbu. Fair and balanced? Bias in bug-fix datasets. In *Foundations of Software Engineering*, pages 121–130, 2009.

[215] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[216] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *Programming Language Design and Implementation*, pages 136–148, 2008.

[217] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

[218] Andrea Arcuri and Gordeon Fraser. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*, pages 33–47, 2011.

[219] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, pages 3–13, 2012.

[220] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416 – 2430, 2010.

[221] Stephanie Forrest, Westley Weimer, ThanhVu Nguyen, and Claire Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference*, pages 947–954, 2009.

[222] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.

[223] Mark Harman. Automated patching techniques: the fix is in (technical perspective). *Communications of the ACM*, 53(5):108, May 2010.

[224] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM Research Highlight*, 53(5):109–116, May 2010.

[225] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Representations and operators for improving evolutionary software repair. In *Genetic and Evoluationary Computation Conference*, pages 959–966, 2012.