Eric Schulte

Candidate

Computer Science

Department

This dissertation is approved, and it is acceptable in quality and form for publication:

Approved by the Dissertation Committee:

Stephanie Forrest, Chairperson

Westley Weimer

Jedidiah Crandall

Melanie Moses

Neutral Networks of Real-World Programs and their Application to Automated Software Evolution

by

Eric Schulte

B.A., Mathematics, Kenyon College, 2004

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

July 2014

Neutral Networks of Real-World Programs and their Application to Automated Software Evolution

by

Eric Schulte

B.A., Mathematics, Kenyon College, 2004

Ph.D., Computer Science, University of New Mexico, 2014

Abstract

The existing software development ecosystem is the product of evolutionary forces, and consequently real-world software is amenable to improvement through automated evolutionary techniques. This dissertation presents empirical evidence that software is inherently robust to small randomized program transformations, or *mutations*. Simple and general mutation operations are demonstrated that can be applied to software source code, compiled assembler code, or directly to binary executables. These mutations often generate variants of working programs that differ significantly from the original, yet remain fully functional. Applying successive mutations to the same software program uncovers large *neutral networks* of fully functional variants of real-world software projects.

These properties of *mutational robustness* and the corresponding *neutral networks* have been studied extensively in biology and are believed to be related to the capacity for unsupervised evolution and adaptation. As in biological systems, mutational robustness and neutral networks in software systems enable automated evolution.

The dissertation presents several applications that leverage software neutral networks to automate common software development and maintenance tasks. Neutral networks are explored to generate diverse implementations of software for improving runtime security and for proactively repairing latent bugs. Next, a technique is introduced for automatically repairing bugs in the assembler and executables compiled from off-the-shelf software. As demonstration, a proprietary executable is manipulated to patch security vulnerabilities without access to source code or any aid from the software vendor. Finally, software neutral networks are leveraged to optimize complex nonfunctional runtime properties. This optimization technique is used to reduce the energy consumption of the popular PARSEC benchmark applications by 20% as compared to the best available public domain compiler optimizations.

The applications presented herein apply evolutionary computation techniques to existing software using common software engineering tools. By enabling evolutionary techniques within the existing software development toolchain, this work is more likely to be of practical benefit to the developers and maintainers of real-world software systems.

Dedication

To my daughter Louisa Jaye, my own personal variant.

Acknowledgments

First and foremost, I would like to thank my advisor Stephanie Forrest. It has been a pleasure to work with someone whose intuition about what is important and interesting align so well with my own. Dr. Forrest's insightful questions led to many of the most fruitful branches of this dissertation.

I am lucky to have worked so closely with Westley Weimer and to have benefited from his dedication to education, technical and domain expertise, and willingness to help with all aspects of graduate school, from questions of collaboration and presentation to the practical aspects of finding a job after graduation. Many thanks to Jed Crandall and Melanie Moses for consistently providing advice and guidance, and for serving on my dissertation committee.

Many generous researchers contributed to this dissertation through review and discussion of the ideas and experiments presented herein. I would especially like to thank the following. Mark Harman for discussion of fitness functions and search based software engineering. Jeff Offutt for generously meeting with me to discuss mutation testing. Andreas Wagner, Lauren Ancel Meyers, Peter Schuster and Joanna Masel for providing a biological perspective of the ideas and results presented herein. William B. Langdon and Lee Specter for reviewing mutational robustness from the point of view of evolutionary computation. Benoit Baudry for discussion of neutral variants of software.

In addition I'd like to thank my colleagues in the Adaptive Computation Lab: George Bezerra, Ben Edwards, Stephen Harding, Drew Levin, Vu Nguyen and George Stelle. It was a pleasure discussing, and often disagreeing, with you about natural and computational systems. Your ideas and critiques helped to flesh out many of the better ideas in this work and to round off many of its rougher edges.

Finally, this work would not have been possible without the inexhaustible support of my partner Christine. Thank you most of all.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We present empirical studies of the effects of small randomized transformations, or *mutations*, on a variety of real-world software. We find that software functionality is inherently robust to mutation. By successively applying mutations we explore large *neutral networks* of fully functional variants of existing software projects. We leverage *software mutational robustness* and the resulting *neutral networks* to improve software through evolutionary processes of stochastic modification and fitness evaluation that mimic natural selection. These methods are implemented using familiar tools from existing software development toolchains, and yield several techniques for software maintenance and improvement which are practical, widely applicable, and easily integrated into existing software development environments.

We demonstrate multiple applications of evolutionary techniques to the improvement of real-world software, including automated techniques to repair bugs in off-the-shelf software and patch exploits in closed source binaries, techniques to generate diverse implementations of a software specification, and methods to optimize complex runtime properties of software. By combining the techniques of software engineering and evolutionary computation this dissertation advances a shared research objective of both fields: automating software development.

We propose that the existing software development ecosystem is the product of evolutionary forces, and is thus amenable to improvement through automated evolutionary techniques. We present empirical evidence of *mutational robustness* and *neutral networks* in software—both of which are hallmarks of evolution previously only studied in biological systems. We demonstrate multiple applications of evolutionary techniques for software improvement. The remainder of this section motivates the work, highlights the main empirical results and practical applications, outlines the organization of the remainder of the document, and provides instructions for reproducing the experiments presented herein.

Motivation: Over the last fifty years the production and maintenance of software has emerged as an important global industry, consuming the efforts of 1.3 million software developers in the United States alone in 2008 [25]. This is projected to increase by 21%, or to over 1.5 million by 2018 [25]. As it stands now, software development is difficult, expensive, and error prone. Software bugs cost as much as $312 billion per year [20], patching a single security vulnerability can cost millions of dollars [152], and while data centers are estimated to have consumed over 1% of total global electricity usage in 2010 [88], no existing mainstream compilers offer optimizations designed specifically to reduce energy consumption.

Empirical Results: We find that existing software projects, when manipulated using common software engineering tools, exhibit properties such as mutational robustness and large neutral networks which have previously been studied only in biological systems. In such biological systems mutational robustness and neutral networks have been shown to enable evolutionary processes to function effectively. Our results suggest that the same is true of software systems.

Applications Evolutionary computation improves the fitness, or performance, of a population of candidate solutions through an iterative cycle of random modification, evaluation, and selection in a process greatly resembling natural selection [63, 64]. Evolutionary computation techniques typically begin with an initial population of

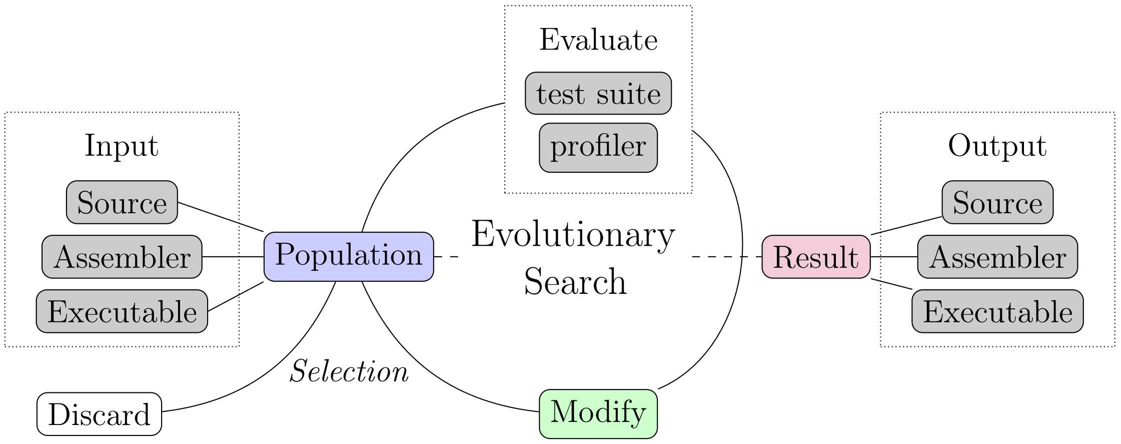


Figure 1.1: Improving real-world software by applying evolutionary computation techniques which are implemented using software engineering tooling.

randomly generated individuals. The methods of representation and the transformation operators of candidate solutions are often domain-specific and selected for the particular task at hand [130].

This work improves an existing instance of software that is already close to optimal.[1] Instead of using domain-specific representations and transformations (as in most prior work in genetic programming), existing software engineering tools are used to ensure applicability to the wide range of complex software used in the real world as well as interoperability with existing software development environments. Building on prior work using genetic programming to repair bugs in existing software [161, 96], we adopt evolutionary computation techniques to support the software engineering applications presented in later chapters.

Organization We present empirical results suggesting that today's software engineering toolchain supports the operations required by evolutionary computation techniques (Chapter 3; including previously unpublished work and work published in Genetic Programming and Evolvable Machines (GPEM) [144]), and more specifi-

[1]The techniques studied herein more closely resemble evolution in natural systems than traditional applications of evolutionary computation do. Biological systems do not create new organisms from whole cloth but rather improve extant organisms.

cally, the program representation, transformation operators, and evaluation functions (Sections 3.1 and 3.2; including previously unpublished work and work published in Automated Software Engineering (ASE) 2010 [143]). We show that software systems have high mutational robustness and large neutral networks, which makes them amenable to improvement through evolutionary processes (Section 3.3).

We next apply evolutionary techniques to a number of software engineering tasks such as repairing bugs in off-the-shelf software (Chapter 5; published in Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2013 [141]), patching vulnerabilities in a closed-source binary (Chapter 6; previously unpublished), generating diverse implementations of a specification (Chapter 4; published in GPEM [144]), and reducing the energy consumption of popular benchmark programs (Chapter 7; published in ASPLOS 2014 [142]). We conclude with a discussion of the outstanding challenges standing in the way of wider adoption of these techniques, possible steps to overcome these challenges, and a summary of the impact of this work to date (Section 8).

We review the relevant literature (Chapter 2) and define the technical terms used in this work (Glossary).

Reproducibility The tools, data, and instructions required to reproduce the experimental results presented in this work are provided. Appendix A describes the software developed for this research and includes pointers to supporting libraries and the implementations of applications. Appendix B describes the data sets used in the experiments, including multiple suites of benchmark programs. In some cases the analysis that produced Figures and Tables is preserved in Org-mode files [41], to support automated reproducibility [140], and links to these files are provided as footnotes.

Chapter 2

Background and Literature Review

This research builds upon previous studies of biological and computational systems. The study of robustness and evolvability in biology provides the concepts and terminology with which we investigate the same principles in computational systems.

Moving from the biological to the computational, this chapter reviews the most relevant results from prior work in biology (Section 2.1), evolutionary computation (Section 2.2), and software engineering (Section 2.3). The final section of this chapter focuses on the immediate precursor work; the use of evolutionary techniques to automatically repair defects in software source code (Section 2.4).

2.1 Robustness and Evolvability in Biology

The ability of living systems to maintain functionality across a wide range of environments and to adapt to new environments is unmatched by man-made systems. The relationship between robustness and evolvability in living systems has been studied extensively. This section reviews this field of study, highlighting the elements most relevant to this dissertation.

Living systems consist of a *genotype* and a *phenotype*. The *genotype* is the heritable information which specifies, or gives rise to the organism. The resulting organism and its behavior, or interaction with the world, is the *phenotype*.

Both the genetic and phenomenal components have associated types of robustness. Genetic robustness, or mutational robustness, is the ability of a genotype to consistently produce the same phenotype despite perturbations to its genetic material. Genetic robustness can be achieved in different ways and on different levels. At the lowest level, important amino acids are over-represented in the space of possible *codon* encodings of triplets of base pairs, making random changes to encoding more likely to produce useful amino acids. Further, functionally similar amino acids have similar encodings. As a result of these properties, many small mutations in amino acid codings are likely to encode identical or similar amino acids [86], making the organism more robust to mutations affecting amino acid encodings. At higher levels, vital functions are often degenerate, meaning that they are implemented by diverse partially redundant systems [43]. For example, in the nervous system no two neurons are *equivalent*, but no single neuron is *necessary*, resulting in a system whose functionality is robust to small changes. Degenerate systems may be more evolvable than systems which achieve robustness through mere redundancy [48, 163]. Finally, many mechanisms have evolved that *buffer* environmental changes, e.g., metabolic pathways whose outputs are stable over a wide range of inputs [157].

Environmental robustness is the ability of a phenotype to maintain functionality despite environmental perturbations. Many of the biological mechanisms responsible for environmental robustness improve the overall robustness of the organism and as a result contribute to mutational robustness [98]. There is a strong correlation between genetic and environmental robustness [84].

Mutational robustness appears to be an evolved feature because evolution tends to increase the mutational robustness of important biological components [31, 164]. Although it is unlikely that mutational robustness is explicitly favored by natural selection as a protection against mutation (due to the low mutation rates in most populations), it seems likely that it arose as a side-effect of evolution to enhance environmental robustness [153].

In evolutionary biology *fitness* is the measurement of an organism's ability to survive and reproduce. *Fitness landscapes* are used to visualize the fitness of related genotypes [165]. Typically one dimension of a fitness landscape encodes an organism's fitness as a scalar, and all other dimensions are used to represent genotypes. This *space* of genotypes may be a high-dimensional discrete space in which each point is a genotype and the immediate neighbors of each point are the genotypes that are reachable by application of a single mutation to the original point.

A mutationally robust organism has many genotypes that map to phenotypes with the same fitness [82]. Regions of the space consisting of genotypes with identical fitness are called *neutral spaces* [81], or neutral networks, and are depicted in Figure 2.1.

Figure 2.1: Neutral spaces are subsets of an organism's fitness landscapes in which every organism has identical fitness.

The relationship between mutational robustness and neutral networks and evolvability is complex [157, 156, 42]. Mutational robustness can be explicitly selected for in static environments and selected against in dynamic environments [164, 111]. Periods of time without selection can increase mutational robustness [153], and periods of strong directional selection can reduce mutational robustness [60].

Mutational robustness directly inhibits evolution by reducing the likelihood that any given genetic modification will have a phenotypic effect. This inhibitory effect can dominate at the small time scales of individual mutations. In some cases overly large neutral networks can actually reduce evolvability [5]. However, large neutral networks predominately promote evolvability. Populations tend to spread out across neutral networks via a process called *drift*, accruing genetic diversity and novel genetic material [30, 145, 18, 99]. This accrued genetic material is believed to play an important role in evolutionary innovation, and provides the genetic fodder required for large evolutionary advances [155, 107, 125, 128, 3].

2.2 Evolutionary Computation

This section describes research into the application of evolutionary techniques in computational systems. We describe the fields of *digital evolution* and *evolutionary algorithms*. In *digital evolution* computational systems are used to perform experiments not yet feasible in biological systems. In *evolutionary algorithms* engineered systems are optimized using algorithms which mimic the biological process of natural selection.

2.2.1 Digital evolution

Some of the research into evolutionary biology discussed in Section 2.1 rely on experiments not based on direct observation of biological systems, but rather on computational models of evolving systems. The evolutionary time frames and the degree of environmental controls required for such experiments are not yet achievable in experiments using biological systems. In digital evolution, much more control and measurement is possible via computational models of evolving populations. These models represent genotypes using specialized assembly languages in environments in which their execution (phenotype) determines their reproductive success [126].

In addition to modeling biological evolution, work in digital evolution has generated hypotheses about those properties of programming languages that might encourage evolvability [124]. Although the languages, or *chemistries*, studied in these computational environments are far from traditional programming languages, some predictions do transfer, such as the brittleness of absolute versus symbolic addressing to reference locations in an assembler genome[124]. Work in the computer virus community has produced similar research on the evolvability of traditional x86 assembly code [69]. One of the contributions of this dissertation is to support some of these claims and intuitions while disproving others. For example, Section 3.3.2.5 confirms the brittleness of absolute addressing in Executable and Linkable Format (ELF) files, while in Chapter 5 we find that x86 assembler (ASM) may be effectively evolved, contrary the conclusions of the computer virus community.

2.2.2 Evolutionary Algorithms

Evolutionary algorithms, including both the *genetic algorithms* and *genetic programming* sub-fields, predate the work on digital evolution.

Genetic algorithms are techniques which apply the Darwinian view of natural selection [36] to engineering optimization problems [63, 113]. Genetic algorithms require a fitness function which the algorithm seeks to maximize. A population of candidate solutions, often represented as a vector, is randomly generated, and then maintained and manipulated through transformations operations typically including mutation and crossover.[1] The algorithm proceeds in a cycle of fitness based selection, transformation, and evaluation, which is usually organized into generations, until a satisfactory solution is found or a runtime budget is exhausted.

Steady state genetic algorithms such as those used in Chapters 6 and 7 are not organized into discrete generations, rather steady state genetic algorithms apply the selection, transformation and evaluation cycle to single individuals [104]. New individuals are immediately inserted into the population and when the size of the population exceeds the maximum allowed population size individuals are selected for eviction. The use of a steady state evolutionary algorithm simplifies the implementation of genetic algorithms by removing the need for explicit handling of generations. Steady state algorithms also reduce the maximum memory overhead to almost 50% of generational techniques because of the interleaved individual creation and eviction. Of most relevance to this work, steady state algorithms are more exploitative than generational genetic algorithms and are thus more appropriate in situations which begin with a highly fit solution, such as in this work which uses evolutionary techniques to improve existing software.

Genetic programming is a specialization of genetic algorithms in which the candidate solutions are programs, often represented as Abstract Syntax Trees(ASTs) [64, 89], and have been applied to a number of real-world problems [130, Chapter 12].

[1]The construction of an initial population "from scratch" is a significant difference between the practice of evolutionary algorithms and the application of natural selection in biological systems, where the starting point of evolutionary processes is always a relatively fit organism (cf. *NK-studies* [128]). The applications presented in the following chapters only work to improve extant software, and more closely resemble natural selection in biological systems.

Genetic programming languages are often much simpler than those used by human programmers and rarely resemble regular programming languages, although in some cases machine code has been evolved directly [91]. Recently genetic programming techniques have been applied to the repair of extant real-world programs [161]. This application is discussed at greater length in Section 2.4.

Genetic algorithms and genetic programming differ both in the way candidate solutions are represented (by using vectors and trees respectively [130, Chapter 2]) and in the application to programs in the case of genetic programming. The work presented in this dissertation applies to existing programs as does genetic programming. However, vector representations are often used, as they are in genetic algorithms. so the blanket term "evolutionary computation" is used in this dissertation to avoid confusion.

The performance of evolutionary algorithms are highly dependent upon the properties of the fitness landscape as defined by their fitness function. *NK-studies* have been used to access the effectiveness of these techniques over tunable fitness landscapes [10, Section B2.7.2]. In these studies the values N and K may be tuned to control the ruggedness of the landscape being searched. N controls the dimensionality of the space, and K controls the epistasis (the degree of interaction between parts) of the space. Chapter 7 in this work explores the use of *smoother* fitness functions than those used in prior chapters. Further discussion of fitness functions are given in Sections 3.6.4 and 8.2.2.

Prior work in genetic programming has leveraged vector program representations applied to machine code. The limited amount of previous work in this field falls into two categories. The first category guarantees that the code-modification operators can produce only valid programs, often through complex processes incorporating domain-specific knowledge of the properties of the machine code being manipulated [119, 127]. In the second approach, the genetic operators are completely general and the task of determining program validity is relegated to the compiler and execution engine [91]. This dissertation takes the latter approach, and we find that both compiler Intermediate Representations(IRs) and ASM are surprisingly robust to naïve modifications (Section 3.3.2.5).

2.3 Software Engineering

This dissertation contributes to the larger trend in software engineering of emphasizing acceptable performance over formal correctness. We review recent work in this vein and highlight tools and methods of particular interest.

Approximate computation encompasses a variety of techniques that seek to explore the trade-off of reduced accuracy in computation for increased efficiency. The main motivation behind this work is the insight that existing computational systems often provide much more accuracy and reliability than is strictly required from the level of hardware up through user-visible results. Examples of promising approximate computation techniques include neural accelerators for efficiently executing delineated portions of software applications [47], and languages for the construction of reliable programs over unreliable hardware [27, 19].

In *failure oblivious* computing [134], common memory errors such as out-of-bounds reads and writes are ignored or handled in ways that are often sufficient to continue operating but not guaranteed to preserve program semantics. For example, a memory read of a position beyond the end of available memory can be handled in a number of different ways. The requested address can be "wrapped" modulo the largest valid memory address. Memory can be represented as a hash table in which addresses are merely keys and new entries are created when needed. In this case reads of uninitialized hash entries can simply return random values. By preventing common errors such as buffer overruns these techniques have been shown to increase the security and reliability of some software systems. Failure oblivious computing assumes that in many cases security and reliability are more important than guaranteed semantics preservation [132, 135, 102, 133].

Beal and Sussman take a similar approach proposing a system for increasing the robustness of software by pre-processing program inputs [14]. Under the assumption that most software operates on only a sparse subset of the possible inputs, they propose a system for replacing aberrant or unexpected inputs with fabricated inputs remembered from previous normal operation. This system of input "hallucination", is shown to improve the robustness of a simple character recognition system.

While the previous system learns and enforces invariants on program input, the clearview system [129] learns invariants from trace data extracted from a running binary using Daikon [46]. When these invariants are violated by an exploit of a vulnerability in the original program the system automatically applies an invariant-preserving patch to the running binary, which ensures continued execution. ClearView was evaluated against a hostile red-team and was able to successfully repair seven out of ten of the red team's attacks [129]. Despite these impressive results the ClearView system has a number of limitations. The tool used to collect invariants (Daikon) is not exhaustive (e.g., missing polynomial and array invariants [118]), is only able to detect a limited set of errors, is only able to repair a pre-configured set of errors for which hand written templates exist, and is not guaranteed to preserve correct program behavior.

The approaches mentioned above are applied to executing software systems. There are also techniques that apply to the pre-compiled software source-code, or genotype. One family of such techniques includes *loop-perforation* [112] and *dynamic knobs* [62]. In loop-perforation, software is compiled to a simple IR, looping constructs are found in this IR and then modified to execute the loop fewer times by skipping some loop executions. This technique can be used to reduce energy and runtime costs of software while maintaining probabilistic bounds of expected correctness. This work is notable for introducing program transformations that are not formally semantics preserving but are rather predictably probabilistically accurate. The impact of these techniques is limited to specific manually specified program transformations, a more general method of program optimization, of which loop perforation may be a special case, is given Chapter 7.

There is a common misconception that software is brittle and that the smallest changes in working code can lead to catastrophic changes in behavior. This perceived fragility is codified in *mutation testing* systems. Such systems measure program test suite coverage by the percentage of random program changes that cause the test suite to fail, and operate under the assumption that random changes to working programs result in breakage [101, 39, 66, 74]. This usage presumes that all program mutants are either buggy or equivalent to the original program.

The detection of equivalent programs is a significant open problem for the mutation testing community (cf. *equivalent mutant problem* [74, Section II.C]) [58]. Although the detection of equivalent mutants is undecidable [24], a number of tools have been proposed for automatically finding equivalent mutants [120, 123, 138]. By contrast, the techniques presented herein exploit neutral and beneficial variation in program mutants for use in software development and maintenance. Recent work by Weimer *et al.* [160], seeks to leverage work on equivalent mutants from the mutation testing community to improve the efficiency of automated of bug repair.

2.4 Genprog: Evolutionary Program Repair

Genprog is a tool for automatically repairing defects in off-the-shelf software using an evolutionary algorithm. It does not require a formal specification, program annotations, or any special constructs or coding practices. It requires only that the program be written in C and be accompanied by a test suite [161, 96].

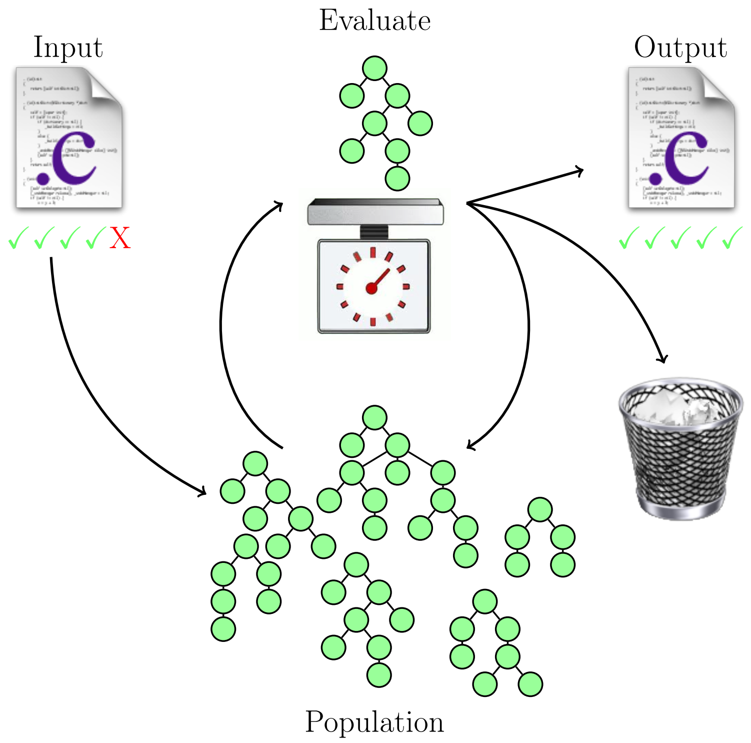


Figure 2.2: Genprog automated evolutionary program repair (Le Goues [94, Figure 3.2]). Input includes a C program, a passing regression test suite, and at least one failing test indicating a defect in the original program. The C source is parsed into an AST which is iteratively mutated and evaluated. When a variant of the original program is found which continues to pass the regression test suite and also passes the originally failing test, this variant is returned as the "repair."

The Genprog repair process is shown in Figure 2.2. As input Genprog requires the C source code of the buggy software, the regression test suite which the current version of the software is able to pass, and at least one failing test indicating the bug. The source is parsed into a C Intermediate Language (CIL) AST [116, 161], which is then duplicated and transformed using the three mutation operations (shown in Panels a-c of Figure 3.2) and crossover to form a population of program variants.

In an evolutionary process of program modification and evaluation, Genprog searches for a variant of the original program which is able to pass the originally failing test case, while still passing the regression test suite. This version is returned by the evolutionary search portion of the Genprog technique. As a final post-processing step the difference between the original program and the repair is minimized to the smallest set of diff hunks required to achieve the repair. This minimization is performed using *delta debugging*, a systematic method of minimization while retaining a desired property [166].

In a large-scale systematic study Genprog was able to fix 55 of 108 bugs taken from a number of popular open-source projects. When performed on the Amazon EC2 cloud computing infrastructure each repair cost less than \$8 on average [95] which is significantly cheaper than the average cost of manual bug repair.

Genprog has had a significant impact on the software engineering research community. The project won best paper awards at the International Conference on Software Engineering (ICSE) in 2009, the conference on Genetic and Evolutionary Computation (GECCO) in 2009, and Search-Based Software Testing (SBST) in 2009. In addition it earned *Humies* awards for human-competitive results produced by evolutionary algorithms. In 2009 Genprog [161] and ClearView [129] demonstrated the applicability of *automated program repair* to real-world software defects, and Orlov and Sipper demonstrated a technique similar to Genprog over smaller Java programs [127]. Since then interest in the field has grown with multiple applications (e.g., AutoFix-E [158], AFix [75]) and an entire section of ICSE 2013 (e.g., SemFix and FoREnSiC [117, 87], ARMOR [28], PAR [79, 114]).

This dissertation is also a descendant of Genprog. The experiments described in the following chapters investigate the mechanisms underlying Genprog's success (Chapter 3 [144]) and extend the Genprog technique into new areas such as repair in embedded systems (Chapter 5 [141]) repair of closed source binaries (Chapter 6), and optimization to reduce energy consumption (Chapter 7 [142]).

Chapter 3

Software Representation, Mutation, and Neutral Networks

In biological systems both mutational robustness and the large neutral networks reachable through fitness-preserving mutation are thought to be necessary enablers of evolutionary improvement as discussed in Section 2.1. This chapter describes a series of experiments which demonstrate the prevalence of mutational robustness and large neutral networks in software using the existing development toolchain.

Mutational robustness and neutral networks arise in systems with a genetic structure which support mutation and crossover and which give rise to a phenotype that in turn supports fitness evaluation. This chapter defines multiple program representations and the mutation and crossover transformations they support (Section 3.1). These representations include high-level Abstract Syntax Trees(ASTs), compiler Intermediate Representation (IR), assembler (ASM), and binary executables. Methods of expressing these representations as executable programs and evaluating their fitness are given (Section 3.2).

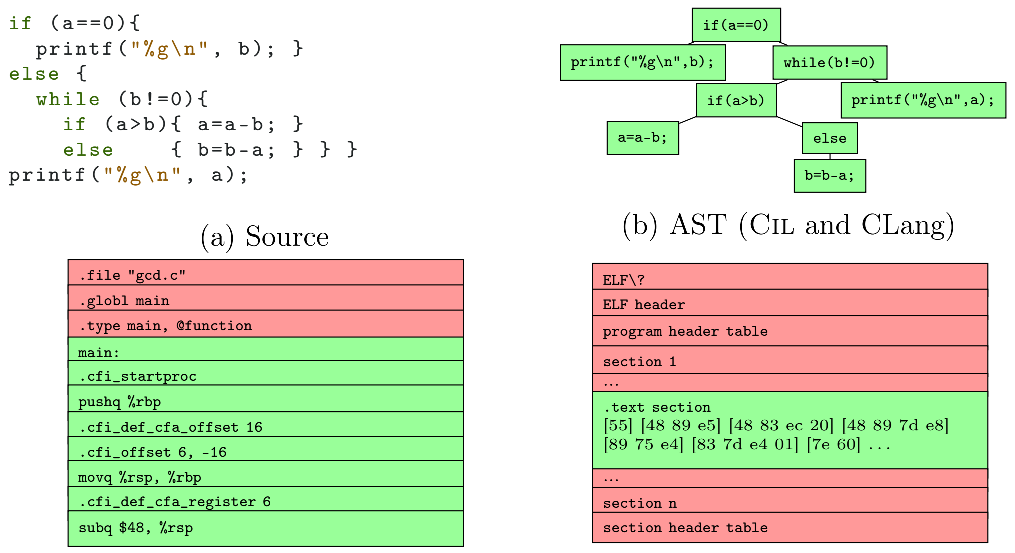
Software mutational robustness is present in real-world programs across multiple levels of representation, regardless of the quality of the method of fitness evaluation (Section 3.3; published in GPEM [144]). Large neutral networks are found and their properties are investigated (Section 3.4; previously unpublished work and work published in GPEM [144]). This chapter concludes with a mathematical analysis of the fitness landscape defined by these program representations and of the resultant neutral networks (Section 3.5; previously unpublished).

3.1 Representation and Transformation

3.1.1 Representations

In this dissertation we view a program's representation as its genetic information, which can be modified with mutation and crossover operations. As discussed in Section 2.1, program representations are typically hierarchical trees linear vectors. In this dissertation we study both forms. By focusing on program representations which are closely based upon structures used commonly in software engineering the implementations of many of these program representations are able to leverage existing tools.

Specifically we will investigate two high-level tree program representations and three low-level vector program representations. The tree program representations include one based on Cil [96], and one based on C Language family frontend for LLVM (CLang) ASTs [92]. The three lower-level program representations include one based on argumented ASM code [143], one based upon Low Level Virtual Machine (LLVM) IR [93], and one program representation applicable directly to binary ELF files [141]. This last representation is operationally similar to the ASM representation with additional bookkeeping required for all program transformations. The



```
if (a==0){
  printf("%g\n", b); }
else {
  while (b!=0){
    if (a>b){ a=a-b; }
    else    { b=b-a; } } }
printf("%g\n", a);
```

(a) Source

(b) AST (CIL and CLang)

```
.file "gcd.c"
.globl main
.type main, @function
main:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $48, %rsp
```

(c) Vector (Assembler and LLVM)

```
ELF\?
ELF header
program header table
section 1
...
.text section
[55] [48 89 e5] [48 83 ec 20] [48 89 7d e8]
[89 75 e4] [83 7d e4 01] [7e 60] ...
...
section n
section header table
```

(d) ELF

Figure 3.1: Program representations. The Cil and Clang tree representations are shown in Panel (a). Panels (c) and (d) shown the ASM, LLVM, and ELF program representations. The source code shown in Panel (a) is not used as a program representation because of the lack of directly source-code level transformations (most tools for source code transformation first parse the source into an AST which is then transformed and serialized back to source).

five program representations are depicted graphically in Figure 3.1 and are described in greater detail below.

3.1.1.1 CLang-AST

The highest level representation is based on ASTs parsed from C-family languages using the C Language family frontend for LLVM (CLang) tooling. This level of representation most closely matches source code written directly by human software developers. This level of representation is used in Section 3.3.2.5.

3.1.1.2 Cil-AST

The next highest level of representation is based on ASTs parsed from C source code using the C Intermediate Language (Cil) toolkit to parse, manipulate and finally serialize C source ASTs back to C source code [116, 161].

Cil simplifies some C source constructs to facilitate programmatic manipulation. Despite these simplifications, the Cil AST representation more closely resembles a high-level source code than a true compiler IR. This level of representation is used in Sections 3.3 and 3.4 and in Chapters 4 and 5.

3.1.1.3 LLVM

The Low Level Virtual Machine (LLVM) representation operates over the LLVM compiler IR, which is written in Static Single Assignment (SSA) form [93]. LLVM supports multiple language front-ends making it applicable to a wide range of software projects.

A rich suite of tools is emerging around the LLVM infrastructure.[1] This representation benefits from these tools because they can be easily applied to LLVM program representations to implement fitness evaluation or program transformation. This level of representation is used in Section 3.3.2.5.

3.1.1.4 ASM

Any compiled language is amenable to modification at the assembler (ASM) level. This level represents programs as a vector of assembler instructions. Some compilers support this translation directly, e.g., the `-S` flag to the `GCC` compiler causes it to emit a string representation of ASM instructions. For our ASM program represen-

[1]An updated list of related publications is maintained at `http://llvm.org/pubs/`.

tation, we parse this sequence of instructions into a vector of argumented assembly instructions. This parsing is equivalent to splitting the output of the string emitted by `GCC -S` on newline characters. This vector can be manipulated programatically (e.g., by mutation and crossover operations) and serialized back to a text string of assembly instructions [143, 141]. The ASM program representation handles multiple Instruction Set Architectures(ISAs) including both Complex Instruction Set Computer (CISC) ISAs such as x86 and Reduced Instruction Set Computer (RISC) ISAs such as MIPS.

This level of representation is used in Sections 3.3 and 3.4 and in Chapters 5 and 7.

3.1.1.5 ELF

The Executable and Linkable Format (ELF) file format is a common format of compiled and linked library and executable files [151]. When the code in ELF files is executed, it is loaded into memory and translated by the CPU into a series of argumented assembler instructions. Using custom tooling in combination with existing disassemblers such as `objdump` it is possible to modify the sequence of assembler instructions in an ELF file in much the same way as with the ASM program representation [141]. This level of representation is used in Section 3.3.2.5 and 3.4 and in Chapters 5 and 6.

3.1.2 Transformations

Every program representation used in this dissertation supports the same set of three simple mutation transformations (*copy*, *delete* and *swap*) and at least one crossover transformation. These transformations are taken from previous work in the genetic programming community where they have been shown to be powerful enough to evolve novel behavior [130, Section 2.4].

All four program transformations are simple and general. They do not encode any domain knowledge specific to the program material they manipulate. They are applicable to multiple program representations (e.g., AST or ASM), and to multiple languages (e.g., x86 or ARM assembler) without modification. These transformations are plausible analogs of common biological genetic transformations, and are commonly performed by human software developers [80].

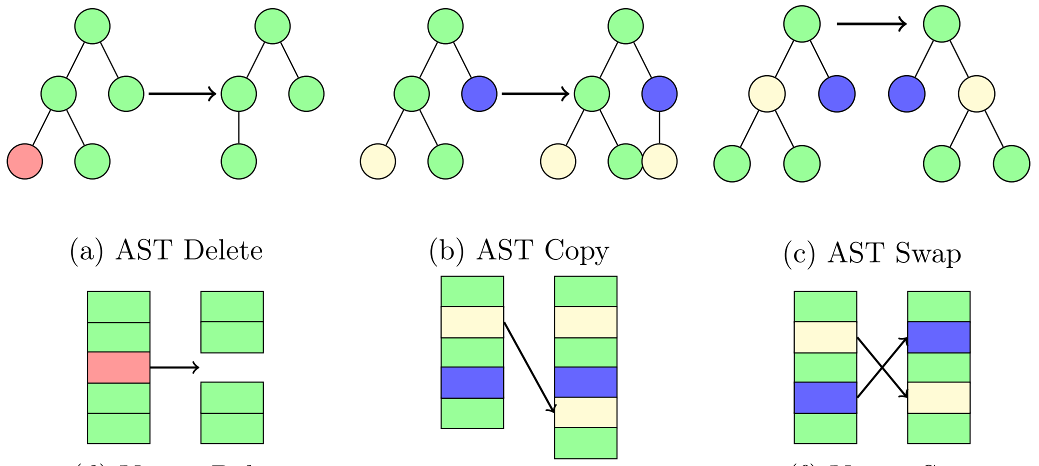
None of these transformations creates new code. Rather they remove, duplicate, or re-order elements already present in the original program. This design is based on the intuition that most extant programs already contain the code required to implement any desirable behavior related to their specification. The benefit of limiting the program transformations in this way is to limit the size of the space of potential programs (Section 3.5).

3.1.2.1 Mutation

The mutation transformations are *copy*, *delete*, and *swap*. *Copy* duplicates an AST subtree, or instruction in vector and inserts it in a random position in the AST or immediately after a randomly chosen location in the vector respectively. *Delete* removes a randomly chosen AST subtree or vector element. *Swap* exchanges two randomly chosen AST subtrees or vector elements.

Figure 3.2 illustrates the mutation operators. ELF mutation operations are similar to ASM mutation operations, are described in greater detail in Section 6.2.2.1, and are shown in Figure 6.2.

The LLVM transformations are more complex than the simple operations shown in Figure 3.2. The LLVM IR requires that a valid data-dependency graph be main-



(a) AST Delete (b) AST Copy (c) AST Swap

(d) Vector Delete (e) Vector Copy (f) Vector Swap

Figure 3.2: Mutation transformations over both tree and vector program representations.

tained by all program transformations. Thus, the LLVM mutations must explicitly patch this dependency graph, assigning inputs and outputs for all new statements and replacing the inputs and outputs of all removed statements. Figure 3.3 illustrates the process for the delete and copy operations. These data dependencies that we manipulate explicitly in the LLVM IR are managed implicitly at the ASM level through the re-use of processor registers.

3.1.2.2 Crossover

Crossover re-combines two program representations and produces two new program representations with elements from each parent in a process analogous to the biological process of the same name. In tree representations, one subtree of each parent is chosen randomly and they are swapped. In the vector representation a two-point crossover is used [37]. First, two indices which are less than the size of the smaller vector are chosen, then the contents of the vectors between these indices are swapped.

Figure 3.4 illustrates both the tree and vector crossover transformations.

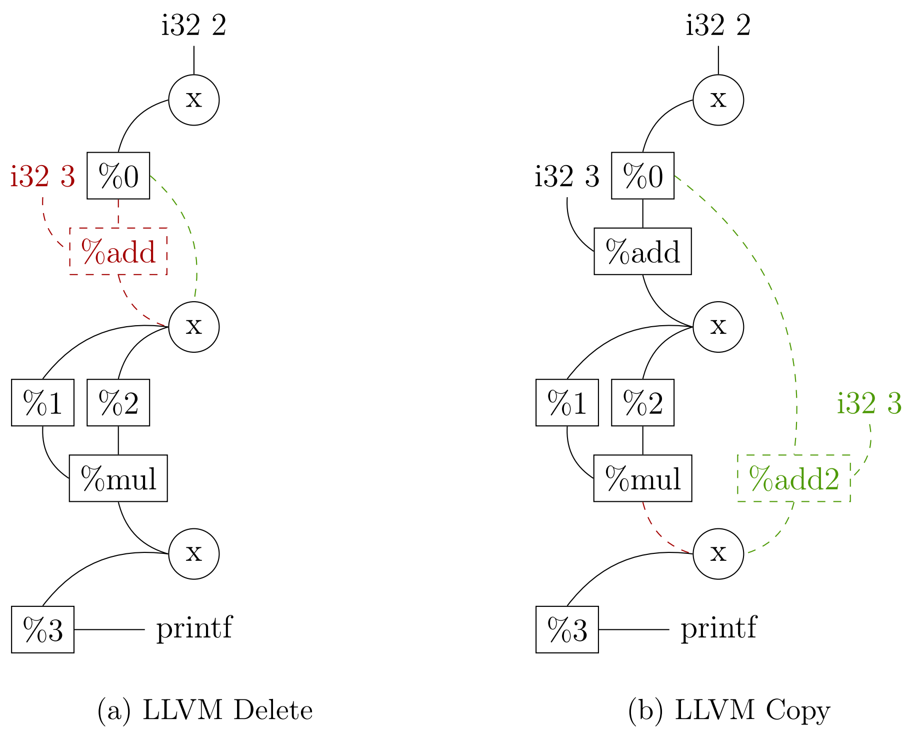


Figure 3.3: Illustration of transformations over LLVM IR. These transformations require that the SSA data dependency graph be repaired after each mutation.

3.1.3 Implementation Requirements

Each level of representation places different requirements on how the software can be programatically manipulated and what parts of the tool chain need to be available.

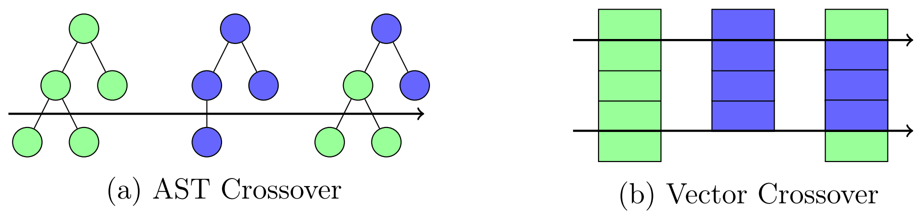


Figure 3.4: Crossover transformations over tree and vector program representations.

AST The CIL-AST and CLang-AST representations have the strictest requirements. Manipulation at the AST level requires that the source code be written in C (for CIL) or a C family language (for CLang) and be available. To express AST programs as executables the entire build toolchain of the software must be available.

ASM and LLVM The ASM and LLVM representations require that the assembler or LLVM IR compiled from the original program be available. To express these programs, the linking portion of a project's build toolchain must be available. Any language whose compiler is capable of emitting and linking intermediate machine code or LLVM IR can use this representation level, making these levels more broadly applicable than the AST representations.

In some cases, re-working a complex software project's build toolchain to emit and re-read intermediate ASM or LLVM IR is not straightforward because not all compilers support such operations directly (e.g., `g++` the C++ front end of the GCC compiler collection).

ELF Any ELF file can be used as input for the ELF program representation. This level of representation does not require access to the source code of the original program. ELF representations can be serialized directly to disk for evaluation and do not require access to the original program's build toolchain. For these reasons the ELF representation is helpful for modifying closed source executables for which no development access or support is available. An example application to a proprietary executable is given in Chapter 6.

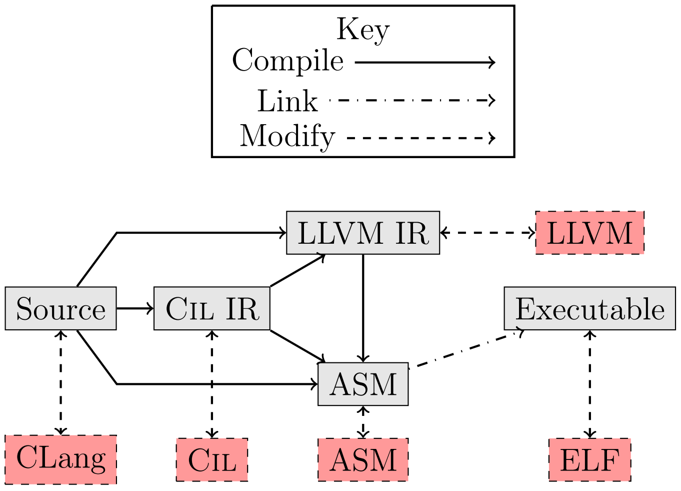


Figure 3.5: Expression of program representations to executables. Gray solid boxes represent traditional software engineering artifacts and red dashed boxes represent program representations. Each program representation modifies a different point in the process of compilation and linking of program code to an executable. In each case only those stages of compilation and linking which are downstream from a program representation are necessary for expression of that program representation.

3.2 Fitness Evaluation

Fitness evaluation in this context is a two-step process. The genotype must first be expressed as an executable (Section 3.2.1), and is then later run against a test suite so that the program phenotype can be evaluated (Section 3.2.2).

3.2.1 Expression

As with biological organisms, a program's fitness is a property of its phenotype or behavior in the world. To assess the fitness of a program, its representation must first be expressed into a phenotype. Expression is shown in Figure 3.5 and varies by program representation as follows.

AST The CIL and CLang AST representations are first serialized back to C-family source code. The CLang tooling produces formatted source code which is identical to the input source. This source code is then compiled and linked into an executable using the build toolchain of the original program.

Expression can fail if the compilation or linking processes fail, for example if an AST violates either type or semantic checks performed by the compiler, or references symbols that cannot be resolved by the linker. In these cases no executable is produced.

ASM The argumented assembler instructions constituting the ASM and LLVM genomes are serialized to a text file. This text code is then linked into an executable using the build toolchain of the original program.

Expression can fail if the linker fails. This is commonly caused by sequences of assembler instructions that are invalid or cannot be resolved by the linker. In this case no executable is produced.

ELF The ELF representation genome is composed of those sections of the ELF file that are loaded into memory during execution. These sections, along with the remainder of the ELF file, are serialized directly into an executable on disk. This process requires no external build tools.

Eliminating the compilation and linking steps for representations at the ASM and ELF levels increases efficiency of evaluation as compared to the other levels. This can dramatic affects the efficiency of techniques using these lower levels (cf. *runtime* Section 5.4).

3.2.2 Evaluation

Evaluation entails executing the program against a test suite. This execution might be evaluated for functional correctness, as in the applications presented in Chapters 4, 5 and 6, or for nonfunctional runtime properties, as in the application presented in Chapter 7.

During both functional and nonfunctional evaluation the programs that fail to express executables are assigned the worst possible fitness value, typically positive or negative infinity.

Functional Evaluation The goal of functional evaluation is to determine if the program behavior is *correct* or *acceptable*. This is determined by its ability to pass all of the test in the test suite. Functional evaluation does not actually determine whether the evaluated program is semantically equivalent to the original program. In many cases a fully functional program variant computes a slightly different function, which is undetectable by the test suite.

Functional evaluation is not related to, nor does it provide any guarantee with respect to, any formal program *specification*. In every case discussed in this dissertation (and in the vast majority of real-world software) there is no written or formal program specification. In these, cases the program's test suite serves as an informal specification. In the cases we present, excluding the repair of proprietary software in Section 6 for which no test suite is available, the test suite distributed with the program is used unaltered.

Nonfunctional Evaluation nonfunctional evaluation assesses the desirability of the nonfunctional runtime properties of a program's execution. Standard profiling tools are used to perform this evaluation. Section 7.1 presents a framework designed to optimize nonfunctional fitness functions. Section 8.2.1 describes additional software engineering tools that could be leveraged in future work to target other types of diverse fitness functions.

3.3 Software Mutational Robustness

Robustness is an important aspect of software engineering research, especially with respect to the reliability and availability of software systems. In contrast to these aforementioned types of phenotypic robustness, this section investigates a form of genotypic robustness which we call *software mutational robustness*. Software mutational robustness refers to the functionality of program *variants*, or instances of software whose genome has been randomly mutated.

Functionality is assessed using the program's test suite as described in Section 3.2.2. We investigate the appropriateness of using program test suites to assess program functionality and find that in the majority of cases test suites serve as a useful proxy for a formal specification. We find that this result holds across a wide range of test-suite qualities—where quality is measured using statement and ASM instruction coverage. Borrowing a term from prior work in biology we call functional program variants *neutral variants*. Neutral variants continue to satisfy the requirements of the original program as defined by the test suite. But, they often differ from the original in minor functional properties such as the order of operations or behaviors left unspecified by the program requirements, and nonfunctional properties may differ from the original, e.g., run-time or memory consumption.

As a simple example, consider the following fragment of a recursive quick-sort implementation.

```
if (right > left) {
  // code elided ...
  quick(left, r);
  quick(l, right);
}
```

Swapping the order of the last two statements like so,

```
quick(l, right);
quick(left, r);
```

or even running the recursive steps in parallel does not change the output of the program, but it does change the program's run-time behavior. We find that examples of such *neutral* changes to programs are commonly and easily discovered through automated random program mutation.

This section provides empirical measurement of software mutational robustness collected across a wide range of real-world software spanning 22 programs comprising over 150,000 lines of code and 23,151 tests. These programs are broken into three broad categories according to the properties of the test-suite. We find an average software mutational robustness of 36.8% and minimum software mutational robustness of 21.2%. We see little variance in software mutational robustness across categories or programs, and we find that the levels of mutational robustness are not explained by the quality of a program's test suite. Given these results we surmise that software is *inherently* mutationally robust.

3.3.1 Experimental Design

This section defines software mutational robustness and describes the techniques used to measure the software mutational robustness of a number of benchmark programs. It will also describe the benchmark programs and their related test suites.

3.3.1.1 Software Mutational Robustness

The formal definition of software mutational robustness is given in Equation 3.1. It is a property of a triplet consisting of a software program P, a set of mutation operations M, and a test suite $T : \mathcal{P} \rightarrow \{\mathsf{true}, \mathsf{false}\}$. *Software mutational robustness* written $MutRB(P, T, M)$ is the fraction of the variants $P' = m(p)$, $\forall m \in M$ for which $t(P)$ (program P passes test t) is true $\forall t \in T$. For any program P, which can not be successfully expressed as an executable $t(P)$ is false $\forall t \in T$, so such programs count in the denominator of $MutRB(P,T,M)$ but not in the numerator.

$$MutRB(P,T,M) = \frac{|\{P' \mid m \in M.\ P' = m(P) \ \wedge \ T(P') = \mathsf{true}\}|}{|\{P' \mid m \in M.\ P' = m(P)\}|} \tag{3.1}$$

3.3.1.2 Measurement of Software Mutational Robustness

In all experiments we use the generic mutation operations described in Section 3.1.2 with equal probability. We evaluate mutational robustness using the CIL-AST and ASM program representations. All of the test suites used are those distributed with the programs P, and are described in more detail along with the description of the benchmark programs in Section 3.3.1.3.

Since exhaustive evaluation of all possible first-order variants (i.e., variants resulting from the application of a single mutation to the original program) is prohibitively expensive (cf. *Number of neighbors* Section 3.5.2), the following technique is used to estimate the mutational robustness of each benchmark program.

1. The original program is run on its test suite and each AST node or ASM instruction executed by the test suite is identified. AST identifying information is collected by instrumenting each AST node to print an identifier during execution. ASM identifying information is collected by using a simple `ptrace`-based utility[2] to collect the values of the program counter during execution. These program counter values are converted into offsets into the program data where they identify specific argumented instructions in the ASM genome.

2. A total of 200 unique variants are generated using each of the three mutation operations for a grand total of 600 unique program variants. Mutation operations are applied uniformly at random along the traces collected in Step 1.

[2]`https://github.com/eschulte/tracer`

Mutations are limited to portions of the programs exercised by the test suite in order to avoid overestimating software mutational robustness by including changes in untested portions of the program.

At the AST level, variants are considered unique if they produce a different assembler when compiled with `gcc -O2`. Non-unique variants are discarded and do not contribute to either the denominator or the numerator of the $MutRB$ fraction to avoid overestimating mutational robustness by counting obviously equivalent mutants as neutral variants. To avoid overestimating mutational robustness, variants which fail to compile are all considered unique and are added to the denominator of the $MutRB$ fraction.

3. Each successfully compiled unique variant is evaluated using the program test suite. Time to execute the test suite is limited to within an order of magnitude of the time taken by the original program to complete the suite. Variants which exceed the time limit (e.g., because of infinite loops) are treated as having failed the test. Only variants that pass every test in the test suite are counted as neutral and added to the numerator of the $MutRB$ fraction.

4. The fraction of unique variants that successfully compile and pass every test in the test suite within the given resource limitations are reported as the software mutational robustness.

3.3.1.3 Benchmark Programs and Test Suites

Our investigation includes 22 real-world software programs listed in Table 3.1. The programs cover three groups including 14 open-source systems programs,[3] four sorting programs,[4] and four programs taken from the Siemens Software-artifact Infrastructure Repository.[5]

[3]`https://cs.unm.edu/~eschulte/repro/robustness.tar.bz2`
[4]`https://github.com/eschulte/sorters`
[5]`http://sir.unl.edu`

The systems programs were chosen to represent the low- to middle-quality test suites which are typical in software development. The test suites used to evaluate the mutational robustness of these programs are those maintained by the software developers and distributed with the programs.

The sorting programs were taken from Rosetta Code,[6] and were selected to be easily testable. We hand-wrote a single test suite to cover all four sorting algorithms.[7] The test suite covers every branch in the AST and exercises every executable assembler instruction in the assembler compiled from the program using `gcc -O2`.

The Siemens programs were selected to represent the best attainable test suites. These programs were created by Siemens Research [67], and later their test suites were extended by Rothermel and Harold until each "executable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 tests" [136]. Also among the Siemens programs, the `space` test suite was created by Volkolos [154] and later enhanced by Graves [57]. The resulting `space` test suite covers every edge in the control flow graph with at least 30 tests.

3.3.2 Results

We report the rate of mutational robustness (Section 3.3.2.1), an analysis of mutational robustness by test suite quality (Section 3.3.2.2), a taxonomy of neutral variants (Section 3.3.2.3), an evaluation of mutational robustness across multiple languages (Section 3.3.2.4) and an evaluation of mutational robustness across all five program representations (Section 3.3.2.5).

[6] `http://rosettacode.org`

[7] `https://github.com/eschulte/sorters/blob/master/bin/test.sh`

3.3.2.1 Mutational Robustness Rates

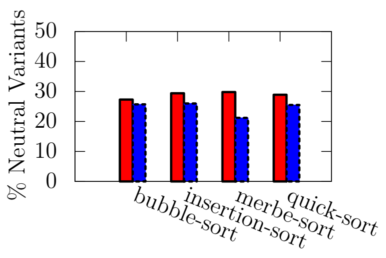
Table 3.1 lists estimated mutational robustness for each of the 22 benchmark programs

Across all programs and representations we find an average mutational robustness of 36.8% and minimum software mutational robustness of 21.2%. These values are much higher than what might be predicted by those who view software as fundamentally fragile. They suggest that for real-world programs there are large numbers of alternate implementations which may easily be discovered through the application of random program mutations. Although actual rates of mutational robustness in biological systems are not available, these rates of software mutational robustness are in the range of mutational robustness thought to support evolution in biological systems [42, Figure 2].

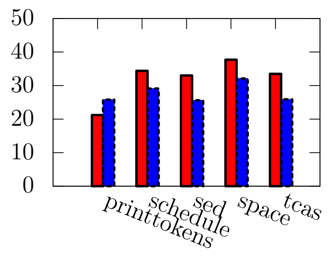
There is little variance in mutational robustness across all software projects despite a large variance in the quality of test suites. At one extreme, even high-quality test suites (such as the Siemens benchmarks, which were explicitly designed to test all execution paths) and test suites with full statement, branch and assembly instruction coverage have over 20% mutational robustness. At the other extreme, a minimal test suite that we designed for bubble sort, which does not check program output but requires only successful compilation and execution without crash, has only 84.8% mutational robustness. This suggests that software mutational robustness is an inherent property of software and is not a direct measurement of the quality of the test suite.

Program	Lines of Code		Test Suite		Mut. Robustness	
	ASM	C	# Tests	% Stmt.	AST	ASM
Sorting						
bubble-sort	184	34	10	100	27.3	25.7
insertion-sort	170	29	10	100	29.4	26.0
merge-sort	233	38	10	100	29.8	21.2
quick-sort	219	38	10	100	28.9	25.5
Siemens [67]†						
printtokens	2419	536	4130	81.7	21.2	25.8
schedule	922	412	2650	94.4	34.4	29.1
space	18098	9126	13494	91.1	37.7	32.1
tcas	544	173	1608	96.2	33.5	25.9
Systems						
bzip2 1.0.2	18756	7000	6	35.9	33.0	26.1
(alt. test suite)			22	71.0	46.4	23.6
ccrypt 1.2	15261	4249	6	29.5	33.0	69.7
(alt. test suite)			16	40.4	34.6	69.7
grep	28776	10929	119	24.9	50.0	36.7
imagemagick 6.5.2	6128	147	145	0.8	33.3	66.3
jansson 1.3	6830	2975	30	28.8	33.3	28.0
leukocyte	40226	7970	5	45.4	33.3	39.9
lighttpd 1.4.15	34165	3829	11	40.1	61.5	56.9
nullhttpd 0.5.0	5951	5575	6	64.5	41.5	37.8
oggenc 1.0.1	299959	59094	10	38.4	33.4	22.1
(alt. test suite)			40	58.8	40.5	72.3
potion 40b5f03	80406	15033	204	48.4	33.3	48.9
redis 1.3.4	44802	17203	234	9.2	33.3	34.0
sed	17026	8059	360	42.0	33.0	25.6
tiff 3.8.2	22458	1732	10	15.4	33.3	90.4
vyquon 335426d	20567	4390	5	50.6	33.3	69.0
total/average	**664100**	**158571**	**23151**	**40.9**	**33.9** ±10	**39.6** ±22

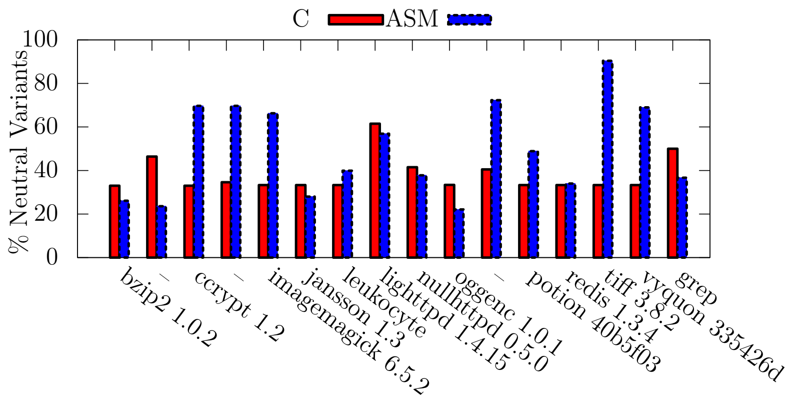
Table 3.1: The mutational robustness of 22 programs spanning three categories. The "Lines of Code" columns report program size in lines of C source code and lines of x86 assembly code. The "Test Suite" columns show the size of the test suite in terms of test cases and the percentage of AST statements exercised by the test suite. The "Mut. Robustness" columns report the rates of software mutational robustness. The ± values in the bottom row indicate one standard deviation. † Although the Siemens benchmark suite claims complete branch and statement coverage, we find less than 100% statement coverage. This is due to our use of finer-grained CIL statements in calculating coverage.



(a) Sorters



(b) Siemens



(c) Systems Programs

Figure 3.6: Mutational robustness by test suite quality. The simple sorting programs in 3.6a have complete AST node and ASM instruction coverage. The Siemens programs in 3.6b have extremely high quality test suites incrementally developed by multiple software testing researchers, including have complete branch and def-use pair coverage. The Systems programs in 3.6c have test suites that vary greatly in quality.

3.3.2.2 Mutational Robustness by Test Suite Quality

Figure 3.6 shows the mutational robustness of the 22 benchmark programs broken into groups by the type and quality of test suite. The differences in test suites are qualitative in the source of the test suites and quantitative in the amount of coverage.

Qualitatively:

Sorting (Panel 3.6a) The sorting test suites all share a single test suite. This test suite leverages the simplicity of sorting program specification and implementations to provide complete coverage with only ten tests.

Siemens (Panel 3.6b) The Siemens test suites were taken from the testing community where they have been developed by multiple parties across multiple publications until each executable statement, and definition-use pair was exercised by at least 30 tests [67, 136].

Systems (Panel 3.6c) The systems test suites are taken directly from real-world open-source projects. These test suites are those used by the software developers to control their own development and consequently reflect the wide range of test suites used in practice.

Quantitatively the sorting test suite provides 100% code coverage at both the AST node and ASM instruction levels. The Siemens programs provide near 100% coverage and the systems programs provide 37.63% coverage on average, with a large standard deviation of 19.34%.

Despite the large difference in provenance and quality, the mutational robustness between panels differs by relatively little as shown in Table 3.2. The average values are within 17% of each other, and the minimums of each group differ by less than 1%.

	Sorting	Siemens	Systems
Average Mut. Robustness	26.7%	29.8%	43.7%
Minimum Mut. Robustness	21.2%	21.2%	22.1%

Table 3.2: Average and minimum mutational robustness of benchmark programs by group.

3.3.2.3 Taxonomy of Neutral Variants

To investigate the significance of the differences between neutral variants and the original program we manually investigated and categorized 35 neutral variants of the `bubble` sorting algorithm. Bubble sort is chosen for ease of manual inspection because of its simplicity of specification and of implementation.

We first generated 35 AST level random neutral variants of bubble sort. The phenotypic traits of these variants were then manually compared to the original program. The results are grouped into a taxonomy of seven categories as shown in Table 3.3.

Number	Category	Frequency
1	Different whitespace in output	12
2	Inconsequential change of internal variables	10
3	Extra or redundant computation	6
4	Equivalent or redundant conditional guard	3
5	Switched to non-explicit return	2
6	Changed code is unreachable	1
7	Removed optimization	1

Table 3.3: A Taxonomy of 35 neutral variants of the bubble sort sorting algorithm.

The categories are listed in decreasing order of frequency. Only categories 1 and some of the variants in category 5 affected the output of the program, either by changing what is printed to `STDOUT` or by changing the final `ERRNO` return value. Both affect program output in ways that are not controlled by the program specification or the test suite.

While some of the remaining five categories affected program output, all but category 6 and some members of category 4 affected the runtime behavior of the program. Category 2 includes the removal of unnecessary variable assignments, re-ordering non-interacting instructions and changing state that is later overwritten or never again read.

Many of the changes, especially in categories 2, 3, and 4, produce programs that will likely be more mutationally robust than the original program. These include changes which insert redundant and occasionally diverse control flow guards (e.g., conditionals that control if statements) as well as changes that introduce redundant variable assignments.

The majority of the neutral variants included in this analysis are semantically distinct from the original program. Only variants in categories 4, 5 and 6 could possibly have no impact on runtime behavior and could be considered semantically equivalent. Instead, the majority of neutral variants appear to be valid implementations of a program specification. In some cases these alternate implementations might have desirable properties such as increased efficiency through the removal of unnecessary code or increased robustness (e.g., the addition of new diverse conditional guards). Subsequent work by Baudry further explores the computation diversity of neutral variants [13], and finds sufficient computational diversity to support *moving target defense* [72].

3.3.2.4 Mutational Robustness across Multiple Languages

To address the question of whether these results depend upon the idiosyncrasies of a particular paradigm, we evaluate the mutational robustness of ASM level programs compiled from four languages spanning three programming paradigms (imperative, object-oriented, and functional). The results are presented in Table 3.4. The uniformity of mutational robustness across languages and paradigms demonstrates that the results do not depend on the particulars of any given programming language.

	C	C++	Haskell	OCaml	Avg.
	Imp.	Imp. & OO	Fun.	Fun. & OO	
bubble	25.7	28.2	27.6	16.7	24.6±5.3
insertion	26.0	42.0	35.6	23.7	31.8±8.5
merge	21.2	46.0	24.9	22.7	28.7±11.6
quick	25.5	42.0	26.3	11.4	26.3±12.5
Avg.	24.6±2.3	39.5±7.8	28.6±4.8	18.6±5.7	27.9±3.1

Table 3.4: Mutational robustness of ASM level programs compiled from four languages spanning three programming paradigms. "Imp." indicates an imperative language, "Fun." indicates a functional language and "OO" indicates an object oriented language.

3.3.2.5 Mutational Robustness across Multiple Representations

This section compares the mutational robustness of the four sorting programs implemented in C across all five program representations. The tests and program implementations used in this section are available online,[8] as well as the code used to run the experiment,[9] and the analysis.[10]

Each of the four sorting algorithms (`bubble`, `insertion`, `merge`, and `quick`) are represented using each of the five program representations (CLang, Cil, LLVM, ASM, and ELF). Each of the resulting 20 program representations is then randomly mutated and evaluated 1000 times.

Table 3.5 shows the average software mutational robustness across all four sorting algorithms broken out by representation. The highest level representation is CLang, which has by far the lowest level of software mutational robustness. The low mutational robustness of the CLang representation is likely an effect of the immaturity of the CLang representation and transformations. It is possible that these trans-

[8] `https://github.com/eschulte/sorters`

[9] `https://github.com/eschulte/sorters/blob/master/src/software-mutational-robustness.lisp`

[10] `https://github.com/eschulte/sorters/blob/master/src/software-mutational-robustness.org`

Representation	Software mutational robustness
CLang	3.12
CIL	20.95
LLVM	35.05
ASM	35.23
ELF	15.78

Table 3.5: Software mutational robustness averaged across four sorting algorithms broken out by each of the five program representations.

formations do not accurately implement the intent of the program transformation operations. For example, an experienced CLang developer would likely produce functionally different *delete* or *copy* transformations of CLang ASTs than those defined in the library used herein.[11]

In general the lower the level of program representation the higher the software mutational robustness with the sole exception of the ELF representation which is fragile due to the need to maintain the overall genome length, and the inability to update literal program offsets.

Figure 3.7 shows the distribution of fitness values across all levels of representation. Each fitness is equal to the number of inputs sorted correctly from a test suite of 10 inputs designed to cover all branches in each sorting implementation. As is the case in biological systems [107], the fitness distribution is bi-modal with one peak at completely unfit variants and another peak at neutral variants. This may help explain the relative stability of software mutational robustness across test suites of varying qualities found in Section 3.3.2.2.

In both biological and computational systems the bi-model fitness distribution may be beneficial, because partially fit solutions are often particularly pernicious (cf. *anti-robustness* [38]). Cancer cells in biological systems are not neutral are able to survive and even thrive, and unfit variants in computational systems which are able

[11]`https://github.com/eschulte/clang-mutate`

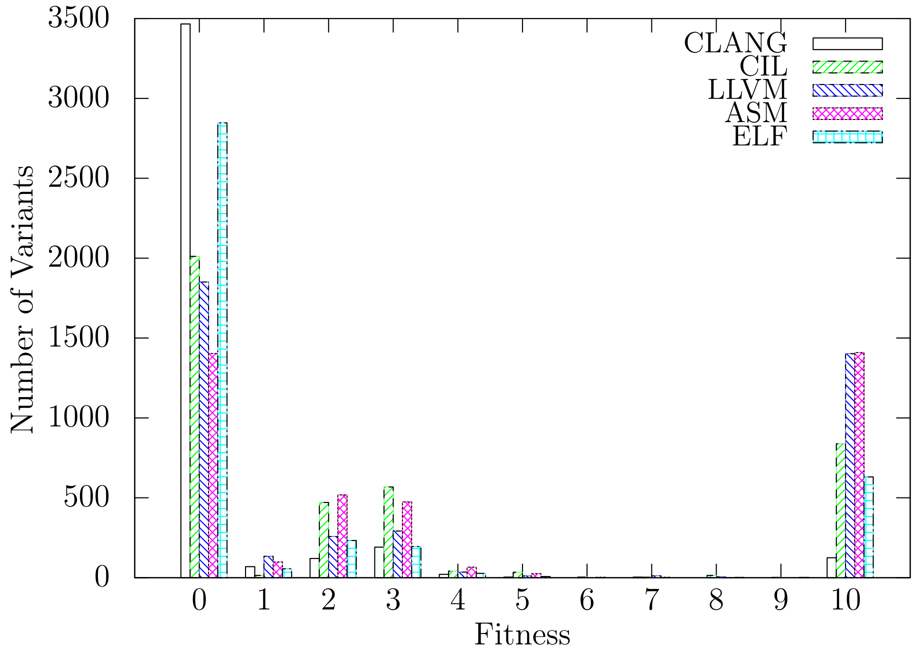


Figure 3.7: Fitness distributions of first order mutations of sorting algorithms by program representation.

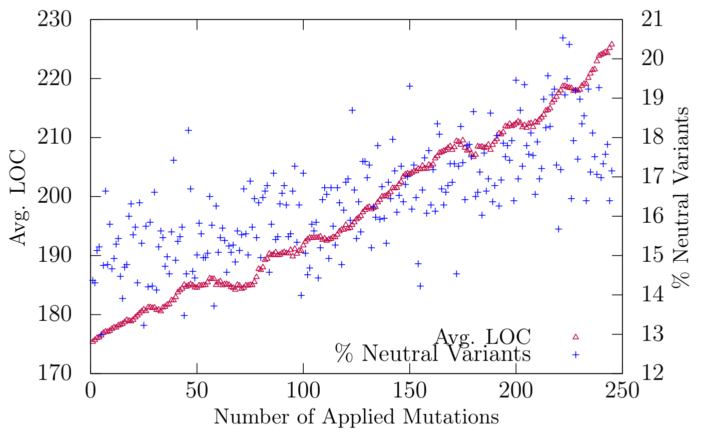
to pass incomplete test suites are the most likely to cause problems for end users. It is not yet clear if similar causes underlay this bi-modal distribution in biological and computational systems.

3.4 Software Neutral Networks

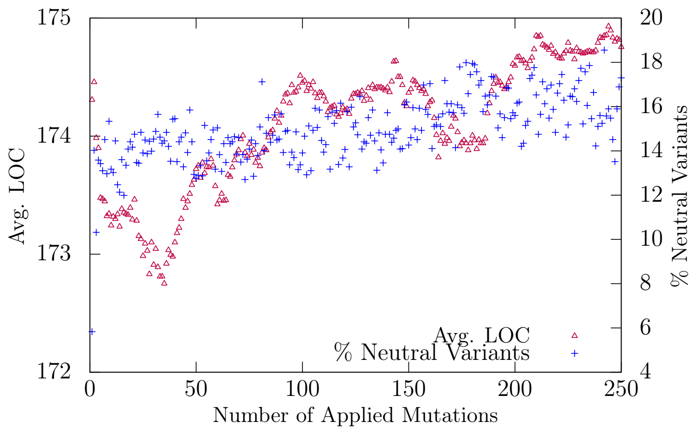
3.4.1 Span of Neutral Networks

The previous experiments measured the percentage of first-order mutations that are neutral. This subsection explores the effects of accumulating successive neutral mutations in a small assembly program. We begin with a working assembly code implementation of insertion sort. We apply random mutations using the ASM representation and mutation operations. After each mutation, the resulting variant is retained if neutral, and discarded otherwise. The process continues until we have collected 100 first-order neutral variants. The mean program length and mutational robustness of the individuals in this population are shown as the leftmost red and blue points respectively in Figure 3.8a. From these 100 neutral variants, we then generate a population of 100 second order neutral variants. This is accomplished by looping through the population of first-order mutants, randomly mutating each individual once retaining the result if it is neutral and discarding it otherwise. Once 100 neutral second-order variants have been accumulated, the procedure is iterated to produce higher-order neutral variants. This process produces neutral populations separated from the original program by successively more neutral mutations. Figure 3.8 shows the results of this process up to 250 steps producing a final population of 100 neutral variants, each of which is 250 neutral mutations away from the original program.

The results show that under this procedure software mutational robustness increases with the mutational distance away from the original program. This is not surprising given that at each step mutationally robust variants are more likely to produce neutral mutants more quickly, and the first 100 neutral mutants generated are included in the subsequent population. We conjecture that this result corresponds to the population drifting away from the perimeter of the program's neutral space. Similar behavior has been described for biological systems, where populations in a constant environment experience a weak evolutionary pressure for increased mutational robustness [153, 157, Chapter 16]. The average size of the program also increases with mutational distance from the original program (Figure 3.8a), suggesting that the program might be achieving robustness by adding "bloat" in the form of useless instructions [130, Section 11.3]. To control for bloat, Figure 3.8b shows the results of an experiment in which only individuals that are the same size or smaller (measured in number of assembly instructions) than the original program are counted as neutral. With this additional criterion, software mutational robustness continues



(a) Program size not controlled.



(b) Program size controlled.

Figure 3.8: Random walk in neutral landscape of ASM variations of Insertion Sort.

to increase but the program size periodically dips and rebounds, never exceeding the size of the original program. The dips are likely consolidation events, where additional instructions are discovered that can be eliminated without harming program functionality.

This result shows that not only are there large neutral spaces surrounding any given program implementation (in this instance, permitting neutral variants as far as 250 edits removed from a well-tested < 200 LOC program), but they are easily traversable through iterative mutation. Section 3.5 analytically explores the possible sizes of these neutral spaces. Figure 3.8b shows a small increase in mutational robustness even when controlling for bloat. Further experimentation will determine if these results generalize to a more diverse set of programs subjects.

3.4.2 Higher Order Neutral Mutants

The previous section demonstrates the ability of automated techniques to explore neutral networks by continually applying single mutations to neutral variants. While this demonstrates the span of neutral networks far from an original program, it does not address questions of the density of neutral variants in the space of all possible programs (cf. *program space* Section 3.5).

To address this question we apply multiple compounding random mutation operations to an individual without performing intervening checks for neutrality. Such *higher-order* random variants take random walks away from the original program in the space of all possible programs. We then evaluate the percentage of these higher order variants at different distances from the original program to see how the rate of neutral variants changes with mutational distance.

We compare the rate of neutral variants along random walks to the rate of neutral variants found along neutral walks which use the fitness function to ensure the walk remains within the program's neutral network as in Section 3.4.1. This empirically confirms the utility of fitness functions in the search for neutral variants, as opposed to suggested alternatives such as random search [59].

Finally we manually examine some interesting higher-order neutral variants which have non-neutral ancestors, and discuss the implication of the low rate of such variants discovered through random search.

3.4.2.1 Experimental Design

Using the ASM representation, an initial population of 2^{10} is generated of first-order neutral variants of an implementation of quicksort in C.[12] This population is allowed to *drift* (Section 2.1) by iteratively selecting individuals from the population, mutating them and inserting only the neutral results back into the population. The result is a neutral exploration which maintains the population size of 2^{10} until a fitness budget of 2^{18} total fitness evaluations has been exhausted. The fitness and mutational path from the original program is saved for every tested variant including non-neutral variants.

The distribution of the number of individuals tested at each number of mutations from the original during the neutral search is saved. A comparison population of 2^{18} random higher-order variants is generated by repeatedly drawing from the distribution of mutational distances resulting from the neutral search, and for each drawn distance generating a random higher-order variant with the same order, or mutational distance from the original. This results in two collections of 2^{18} variants with the same distribution of mutational distances from the original program, one generated through neutral search and the other through random walks. These distributions are shown in Figure 3.9.

The software and input data used to perform this experiment is available online.[13] The analysis is also available online.[14]

3.4.2.2 Neutrality of Random Higher-Order Neutral Variants

As shown in Figure 3.10 the experimental levels of neutrality at two, three, and four random mutations removed from the original almost exactly match an exponential

[12]`https://github.com/eschulte/sorters/blob/master/sorters/quick_c.c`
[13]`https://github.com/eschulte/sorters/blob/master/src/horns.lisp`
[14]`https://github.com/eschulte/sorters/blob/master/src/horns.org`

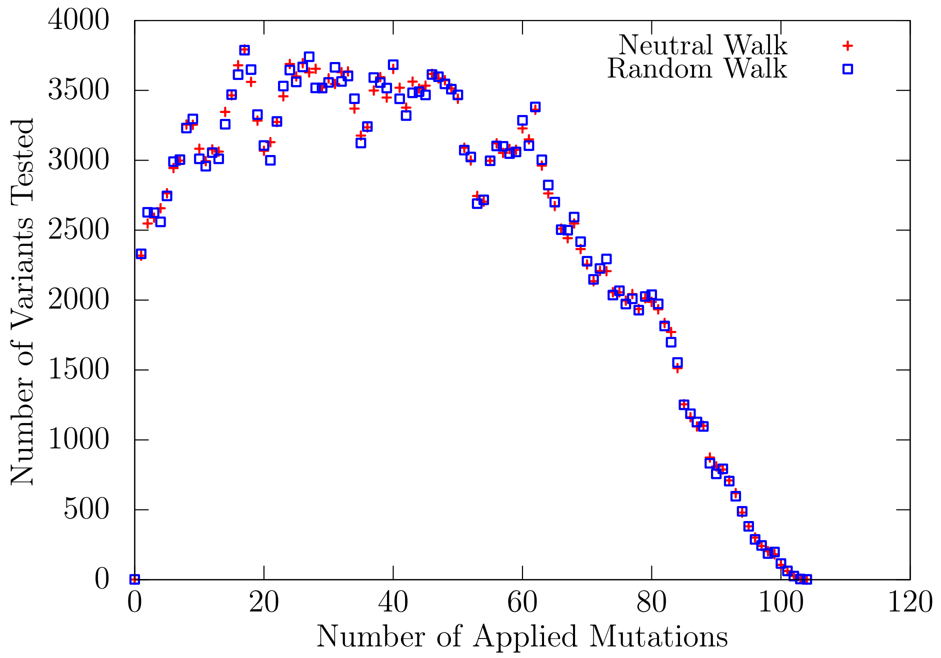


Figure 3.9: The distribution of mutational distance from the original for populations of higher-order variants collected using two methods: random walks and guided neutral walks.

function decreasing at a rate equal to the mutational robustness of the original program. The highest order random neutral variant found in this experiment was 8 mutations removed from the original.

Exponential decay models the number of higher-order neutral variants which will be found through random walks which are neutral at every intermediate step. The very high correlation between exponential decay and the rate of neutrality decrease along random walks indicates that most higher-order neutral variants are the product of neutral lower-order ancestors. This may be an effect of the very high dimensionality of program spaces (cf. *dimensionality* Section 3.5), or of the difference in effective dimensionality between the neutral space and the program space.

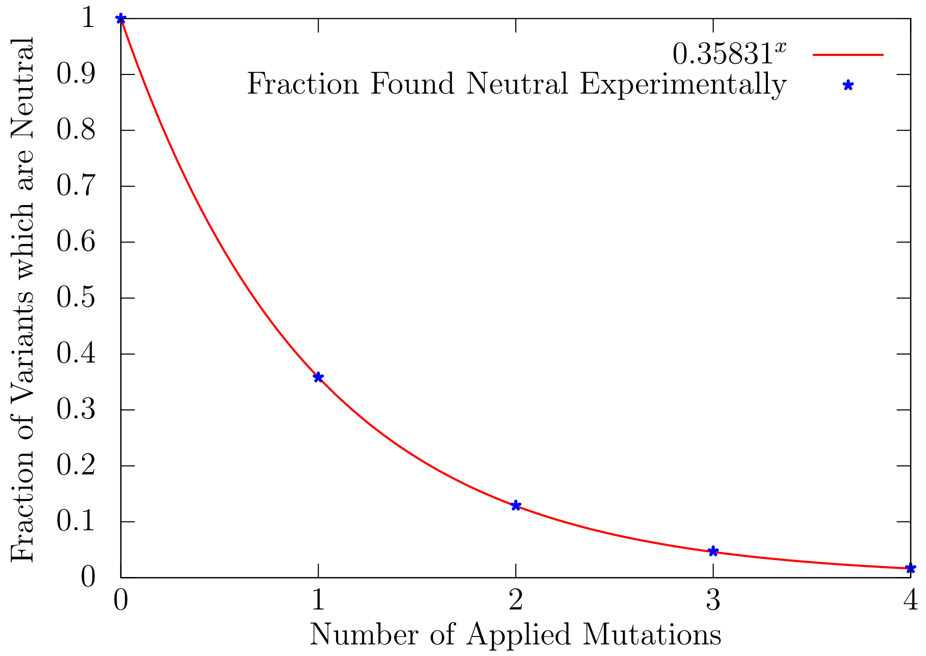


Figure 3.10: Neutrality of random higher-order mutants by the number of mutations from the original.

3.4.2.3 Comparative Neutrality along Random and Guided Walks

Figure 3.11 shows the comparative neutrality by mutational distance up to 100 mutations from the original program for both random and guided walks through program space. By contrast the neutrality *increases* along guided neutral walks. This agrees with the results presented in Section 3.4.1. By the sheer increase in viable higher-order neutral variants found through neutral search as compared to random search, 28,055 and 1,342 respectively, these results support the utility of a fitness function in guiding search for higher-order neutral variants.

3.4.2.4 Analysis of Interesting Random Higher-Order Neutral Variants

Table 3.6 shows the percentage of a large collection of over 50,000 randomly generated higher-order neutral variants which are "interesting," meaning that they have

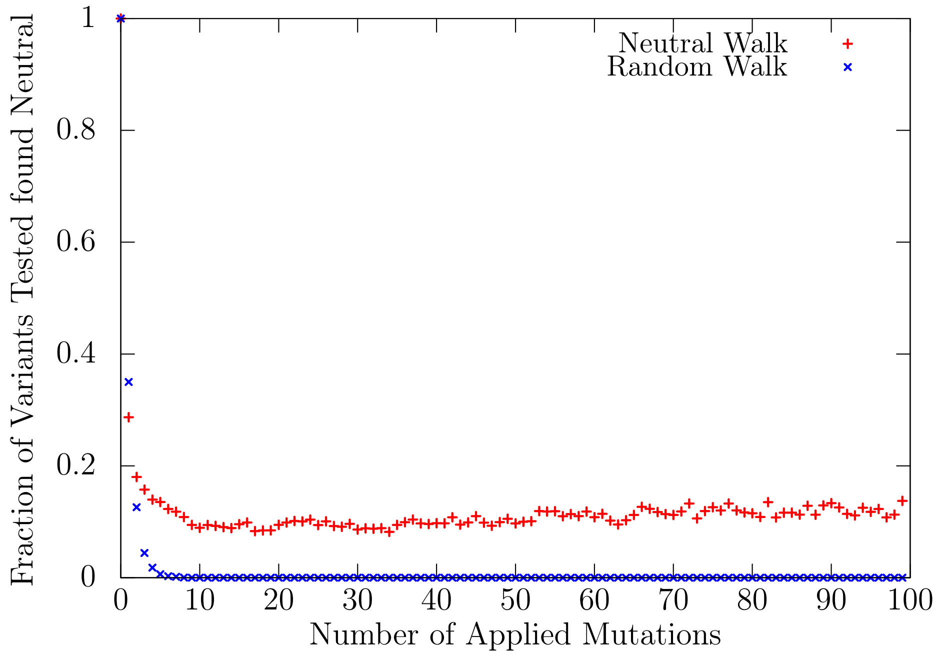


Figure 3.11: The neutrality by mutational distance from the original program for both guided neutral and random walks through program space.

ancestors which were not themselves neutral. These variants are the results of random walks that wander off of the neutral network, and then subsequently wander back on. As can be seen the majority of randomly generated neutral variants are the result of random walks that never leave the neutral network. This agrees with the results found in Figure 3.10.

	Higher-Order Neutral Variants		
Order	Total	Interesting	Percentage
2	33787	139	0.41%
3	12458	153	1.23%
4	4478	107	2.39%
sum	50723	399	0.79%

Table 3.6: Percentage of over 50,000 randomly generated higher-order neutral variants that have non-neutral ancestors.

Note that the percentage of neutral variants which are interesting (column "Percentage" in Table 3.6) increases as the order increases because the number of random intermediate steps increases, each of which has a chance of being non-neutral.

The interesting variants may be further separated into those whose step back onto the neutral network in a reversion of a previous mutation (i.e., a "step back"), and those who return to a different position in the neutral network than the spot of their exit. We find that roughly half $\frac{199}{399}$ are not the result of such a reversion or a "step back". Of these many have fitness histories in which their ancestor's fitness values drop to zero or near zero and then slowly climb or suddenly jump back to neutral.

Such cases of random walks which leave and then re-enter the neutral network in a new place may actually find portions of the neutral network which are unreachable or are very far removed from the original program when restricted to neutral walks. The extreme size of these networks make this possibility difficult to determine experimentally.

Although these results indicate that it is possible to find new (possibly unconnected) portions of the neutral network using random walks, such interesting random walks remain very rare. With 6% of random higher-order variants between 2 and 4 mutations removed from the original being neutral, 0.79% of those being interesting and roughly 50% of those not being reversions we have $0.06 \times 0.0079 \times 0.5 = 0.0002$ or 0.02% of all randomly generated higher-order mutants are truly interesting. This result motivates the technique used in this Chapter and in Chapter 4 of restricting explorations of program space to neutral networks in which functional neutrality is used as the acceptance criteria for every step in the walk.

3.5 Program Spaces and Neutral Network Analysis

This dissertation develops applications for automated software engineering which rely on the discovery of alternate versions of existing programs through application of program transformations to program representations as presented in Section 3.1. Each of these program representations with its associated transformations defines a *program space*. The applications described in Chapters 4 through 7 may be viewed as methods of searching these program spaces. This section provides a mathematical analysis of the size[15] and properties of these program spaces.

The elements of program spaces are program instances, and the distance between any two elements is their edit distance as calculated using the space's mutation operations.[16] Each space has an associated *size*, or the number of different programs which the representation is able to specify. Following the standard practice in analysis of neutral spaces in biology, we set the dimensionality of the space equal to the maximum length of a representation [157, 145], and the number of potential values at each dimension (program spaces are discrete spaces) equal to the number of possible distinct program elements (e.g., AST statements, ASM or LLVM IR instructions). This approach has the undesirable property that the dimensionality of a space is not constant. To keep dimensionality constant, every element of the genome is allowed to take on an $\emptyset$ value indicating no element is present.

[15]We only consider programs of finite length resulting in finite size program spaces. We believe this is a reasonable restriction. Any realistic application leveraging software neutral spaces will place limits on the size of allowable programs as arbitrarily large programs are generally not desirable.

[16]These program spaces may be thought of and effectively modeled as metric spaces. Technically program spaces are not valid metric spaces as not all of the mutation operations are invertible. For example the effect of deleting the last instance of an instruction from an ASM program can not be reversed because there's no other instance remaining in the program to be used by future copy operations. Despite this, the mutation operations can easily be made invertible and cause the program spaces to behave as metric spaces. We do not explore such altered mutation operations in this work because we do not believe the added mutational overhead would be justified by a significant practical effect in mutation operations behavior.

	Cil-AST					ASM				
	Stmts.		Neighbors		Reach	Stmts.		Neighbors		Reach
program	all	uniq.	all	neut.		all	uniq.	all	neut.	
bubble	21	18	609	166.26	$7.1 \times 10^{+26}$	197	138	46689	11999.07	$1.4 \times 10^{+422}$
insertion	20	18	570	167.58	$3.8 \times 10^{+25}$	186	130	41571	10808.46	$6.4 \times 10^{+393}$
merge	29	25	1160	345.68	$1.1 \times 10^{+41}$	237	139	61146	12962.95	$4.2 \times 10^{+508}$
quick	26	23	949	274.26	$7.7 \times 10^{+35}$	230	157	62675	15982.12	$4.9 \times 10^{+505}$
	24.00	21.00	822.00	238.44	$1.7 \times 10^{+32}$	212.50	141.00	53020.25	12938.15	$1.0 \times 10^{+508}$

Table 3.7: Size of neighborhood of sorting programs. Results are given for both the AST and ASM representations. The "Stmts." columns give the number of statements and number of unique statements. The "Neighbors" columns give the number of immediate neighbors reachable by a single mutation, and the expected number of neutral neighbors given the software mutational robustness calculated in Table 3.1. The "Reach." columns give the calculated size of the set of programs reachable from the original program using the given mutation operations.

Table 3.7 presents analysis of the program spaces of the four sorting programs used previously in this Chapter. This analysis includes calculations of neighborhood sizes, calculations of the sizes of neutral networks and sizes of reachable sets. The methods for calculating these values are given in the remainder of this section.

3.5.1 Size of Program Space

Given a program space P with a program length limit of L and a total of U unique elements, the total size of P is $P = (U + 1)^L$. One is added to U to account for removed statements, and the result is raised to the program length because every slot in the program length may take any of $U + 1$ possible values. This holds strictly for the vector representations, but does not take into account differences in possible tree structures at the AST level.[17]

[17]The number of possible binary trees of N nodes is equal to the N$^{\text{th}}$ Catalan Number. Unfortunately calculation of the number of possible tree structures of valid ASTs depends on a number of other factors, for example not all elements of ASTs (only conditionals) can be branch points (and not all elements can live under conditionals, e.g., function definitions can not). The calculations in the remainder of this section are restricted to vector representations.

To develop intuition about the size of these program spaces, we can calculate the size of the space of ASM programs written in MIPS assembler of length $L = 10,00$. Every MIPS instruction is 32 bits long, so there are at most 2^{32} possible argumented MIPS instructions. Including $\emptyset$ statements, there are $2^{32} + 1$ possible values for each element of the program representation. For a 10,000 instruction program this results in a program space which is $10,000^{2^{32}+1}$ elements long. These program spaces are unfathomably large, and it is very likely that only small fractions of these spaces encode useful programs. This motivates the decision in this dissertation to improve upon existing software artifacts rather than attempting to synthesize wholly new instances of software.

3.5.2 Number of Neighbors

The number of neighbors is equal to the number of transformations which generate unique programs. Let $l = length(P)$ be the length of the original program and u be the number of unique elements in the original program. The number of unique neighbors is equal to the number of possible deletes plus the number of possible unique copies plus the number of possible unique swaps. It is impossible that any two different mutation operations (e.g., a swap and an copy) produce the same program, because they each generate a different number of elements in the new program (i.e., $l - 1$ for deletes, l for swaps, and $l + 1$ for copies).

There are l possible delete operations, no two of which result in identical programs.[18] There are roughly $u \times l$ possible copy operations,[19] and $\binom{l}{2}$ possible swap operations so the number of neighbors $N(P) \simeq \binom{l}{2} + lu + 1$.

Table 3.7 shows the size of the neighborhood for each sorting program. Even for very small programs the sizes are quite large with 822 and 53020.25 immediate neighbors for each sorter on average at the AST and ASM levels respectively. Given that these neighborhoods grow exponentially with the length of the program, the number of immediate neighbors will be very large for typical programs of sizes much larger than the sizes of sorters.

In Table 3.1 we empirically measured the rates of software mutational robustness for each sorter. By multiplying the neighborhood size by the rate of software mutational robustness from Table 3.1 we can estimate the total number of first order (i.e., separated from the original program by a single mutation) neutral variants for each sorting program. These estimates are shown in the "Neighbors" columns of Table 3.7.

3.5.3 Reachability

An important property of each level of representation is the fraction of the program space that is reachable from any given starting program, and the length of the shortest edit path to reach that position. Given an initial program p in a program space P, where p has u unique elements and P has e possible elements and a maximum program length of l, the total size of P is $|P| = (e+1)^l$, and the number of programs reachable from p or $R(p)$ to be $R(p) = (u+1)^l$.

[18]Given that delete is represented by overwriting the deleted element with a special *null* element, cases that would normally lead to equivalent programs (e.g., deleting one in a series of identical elements) generate unique program representations.

[19]Technically there, are likely fewer unique copy operations, because copying instruction e_1 either before or after another e_1 instruction produce an identical result. Section 3.5.4 contains more exact calculations of neighborhood in a program space defined by a reduced set of mutation operations.

Every program representation includes inline literals (e.g., 32-bit integers), so any real-world programs p will contain only a small fraction of all possible program elements (e.g., possible C statements at the AST level) or $u < e$. Thus, a search starting from most programs at these levels will cover only very small portions of the total program spaces in which they exist so $R(p) << |P|$.

Leaving literals aside, lower level program representations with smaller instruction sets such as LLVM and CISC ASM, will have much greater coverage of possible program elements. Because of the increased length of most programs in LOC at lower levels (e.g., over a $3\times$ more ASM instructions than AST statements found in Table 5.1), search at the lower levels will often have access to much larger fractions of the total program space and are much less constrained by the original program from which any individual search starts.

3.5.4 Density of Neutral Networks

The question of the total size and density of neutral networks in the program space requires a more analytically tractable model of program space than we've used for our calculations and estimates thus-far in Section 3.5. Namely a space is required where the number of mutation operations leading to identical variants may be determined analytically.

In this section we define such a *rigid* program space based upon the same fixed-length vector program representation, but using a single mutation operator. We calculate the number of new program representations encountered in successive steps through this space. We provide mappings between this rigid space and the space of ASM programs allowing us to project calculations from rigid space to gls:asm program space. Finally we combine these into an expression of the total size of a program's neutral space under the assumption of nearly constant mutational robustness along random walks through the neutral network.

3.5.4.1 Rigid Space

The rigid program space uses the same vector program representation used previous in Section 3.5. Each program is represented as a vector of length d (the dimensionality). In this section the dimensionality will be equal to the length of the original program, so the space will only hold programs of equal or lesser length. This is not a practical limitation as demonstrated in our empirical investigation of the span of neutral networks when program length is limited shown in Figure 3.8b.

The elements which may appear in program vectors are taken from the set E of all possible elements with the set containing the empty set $\emptyset$. Placing $\emptyset$ into a program index is equivalent to deleting the contents of that index. The total size of E is e.

A single mutation operation is defined in this space. This operation is called "*replacement*". Given a variant p, an index $i \leq d$, and an element $e \in E$, replacement sets $p[i] \leftarrow e$. All of the operations defined over vector representations in Section 3.1 may be implemented using only replacement. In addition replacement is ergodic, making this program space a true metric space.

3.5.4.2 Step Wise Expansion in Rigid Space

The number of unique neighbors accessible through successive applications of *replacement* to the original program in this space may be calculated using the following recurrence relation. Note that every variant is reachable from the original program in at most d applications of replacement (or replacement of every index in the program vector).

$$n_{i+1} = \frac{n_i(d-i)(e-1)}{(i+1)} \tag{3.2}$$

The variable d is the dimensionality of the program space and e is the number of elements (including $\emptyset$) which any program dimension may take.

Intuitively, the components of Equation 3.2 have the following meanings

n_i The number of elements of the last expansion. Each element of n_{i+1} will be accessible by a single application of the replacement operation to one of these elements.

$(d - i)$ The number of dimensions at which each element of n_i may be mutated. No member of n_i may be mutated on any of the i previously mutated dimensions and yield a new variant.

$(e - 1)$ The number of new elements which each mutated dimension may take.

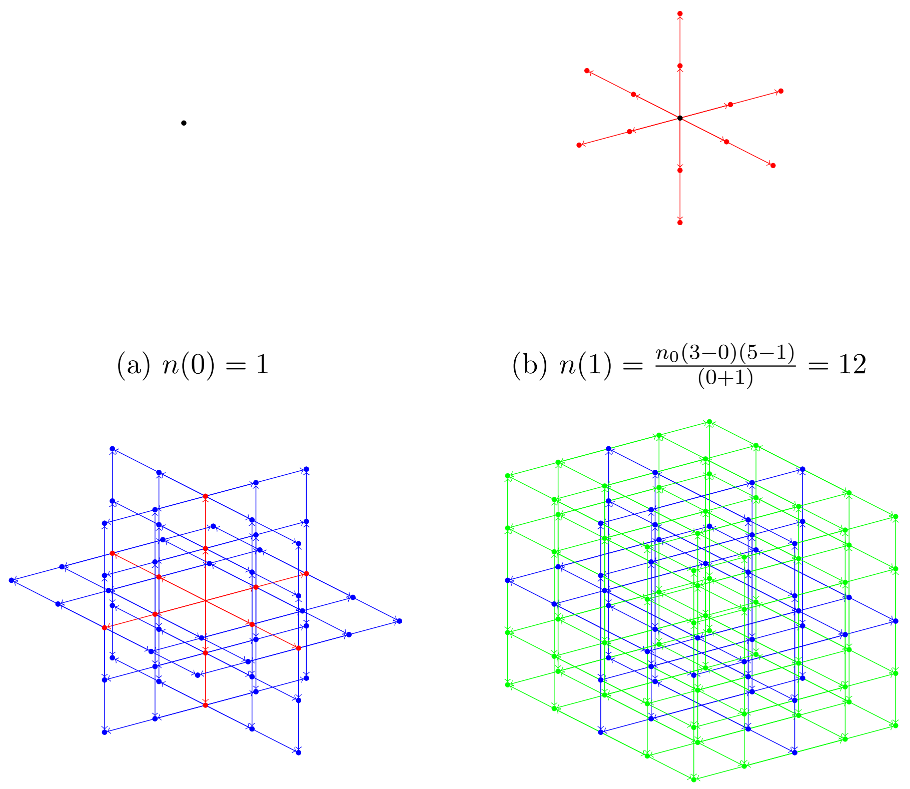
$(i + 1)$ Division by this term accounts for the property of the space that each element of n_{i+1} is reachable from $(i + 1)$ members of n_i.

For every $i \leq d$, the number of variants exactly i applications of replacement from the original program is given by $n(i)$. The value $n(0) = 1$ as only the original program is 0 mutations from the original. Figure 3.12 illustrates these sets of programs in a space in which $d = 3$ and $e = 5$. The original program is placed on the origin for clarity.

The sets $n(i), \forall i \leq d$ are mutually exclusive and partition the program space. The sum of every $n(i)$ is equal to the total number of elements in this space $\sum_{i \leq d} n_i = e^d$.

3.5.4.3 Mappings Between ASM and Rigid Space

There is not a one-to-one mapping from the members of the rigid space defined in this Section to unique ASM programs. Instead, there is a many-to-one mapping with multiple members of this rigid space mapping to the same ASM program. Specifically



(a) $n(0) = 1$

(b) $n(1) = \frac{n_0(3-0)(5-1)}{(0+1)} = 12$

(c) $n(2) = \frac{n_1(3-1)(5-1)}{(1+1)} = 48$

(d) $n(3) = \frac{n_2(3-2)(5-1)}{(2+1)} = 64$

Figure 3.12: The variants accessible through 0, 1, 2, and 3 applications of replacement to an original program in a space with 3 dimensions and 5 possible elements.

this is due to the use of $\emptyset$ elements in this space which are removed to generate the ASM related program.

These $\emptyset$ elements may be placed in any places in rigid space program. Thus the number of rigid space programs which map to any given ASM program is equal to the difference between the length of that program and the length of the original program.

Given an element of ASM of size k, where $k \leq d$ (d is the size of the original) and k+j=d, that element of ASM will have $\binom{l}{j}$ projections into rigid space (equal to the number of slots at which an $\emptyset$ could be placed in the genome). As a result, an element of rigid space with j $\emptyset$ elements will only represent $\binom{l}{j}^{-1}$ of an element in ASM space.

In order to translate the sizes of successive steps in expansion in the rigid space to numbers of programs in ASM, we need to determine the number of elements of each expansion n_i which have z $\emptyset$ elements.

For each expansion i, let $\mathrm{n}_{\mathrm{z,i}}$ be the number of programs with z $\emptyset$ elements in step i. The first program has no $\emptyset$ elements, so $\mathrm{n}_{0,0}$=1. Given this, we know that $\mathrm{n}_{0,1}$ = n_1 - $\mathrm{n}_{1,1}$, and $\mathrm{n}_{1,1}$=d, because there is one variant in n_1 with an $\emptyset$ element for each mutated dimension in the original program.

This leads to the recurrence relation given in Equation 3.3. The number of elements of $n_{z,i+1}$ is composed of elements of $n_{z-1,i}$ which gain an $\emptyset$ in a new dimension, plus elements of $n_{z,i}$ which do not gain an $\emptyset$ in a new dimension. It is not possible for an element of $n_{z+1,i}$ to lose an $\emptyset$, because that would require a repeated mutation in an already mutated dimension. We still have to compensate for duplication, which is accomplished through division by i+1.

$$n_{z,i+1} = \frac{n_{z-1,i}(d-i)1 + n_{z,i}(d-i)(e-2)}{i+1} \tag{3.3}$$

Once we know how many elements of each step have each number of $\emptyset$, we may then find the cardinality of the projection of any given step i into ASM space, A in Equation 3.4.

$$A_i = \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1} \tag{3.4}$$

3.5.4.4 Total Size of Neutral Networks

We assume that a single value of software mutational robustness holds for each step of expansion in the rigid program space. Let r be the rate of software mutational robustness. We may then calculate the total number of neutral variants in ASM. For each $i \leq d$ we may calculate the number of neutral variants in n_i in rigid space. These sizes may then be converted to cardinalities of the ASM programs mapped to by elements of n_i as in Equation 3.5.

$$A_i = r^i \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1} \tag{3.5}$$

The sum of these cardinalities $\forall i \leq d$ will be the total number of neutral variants of the original program in ASM is R in Equation 3.6, with $n_{z,i}$ is calculated using Equation 3.3.

$$R = \sum_{i \leq d} r^i \sum_{z \leq i} n_{z,i} \binom{l}{z}^{-1} \tag{3.6}$$

The expression in Equation 3.6 for the total number of neutral variants of a program in a program space of a given size has a number of important implications. Every possible optimization of the original program (such as those found using the technique described in Chapter 7) will be a member of this large neutral network. Similarly, refinements of the program specification which preserve existing behavior will be constrained to points within this large neutral network.

Previous work by Martinez and Monperrus analyzing program space concludes that program repairs requiring more than ≥ 5 or ≥ 10 steps in program space are impossible to find through automated search [106]. In light of the analytic expressions for the size of neutral spaces presented in Equation 3.6, this prior analysis should be redone to give the chance of finding *any* point in the neutral network of the repaired program, rather than the chance of finding a single unique instance of the repaired program.

3.6 Discussion

The results presented in this chapter contradict the prevailing folk wisdom that software is a precise and intentionally engineered mechanism, which is brittle to small perturbations. We find software to be robust to random mutations, malleable within extensive neutral networks. In this section we discuss a number of implications of this change in perspective.

3.6.1 Evolutionary Provenance of Software

Over the past fifty years software developers have been selecting, reusing, and modifying efficient and robust software development tools, code, and design patterns. Like their biological counterparts, software artifacts which have stood the test of time including applications, interfaces, operating systems, programming languages, utilities, libraries, compilers and linkers all display both robustness and adaptability (in some cases to the exclusion of engineering quality [56]). The history of the existing software development ecosystem can be viewed as an evolutionary process, albeit one in which human engineers are the mechanisms of both mutation and selection [1]. Ultimately, software may stand with biological organisms as a second example of an evolved complex system.

This history of development, through a process mirroring natural selection, has produced the surprisingly biological features of software which were empirically demonstrated in this chapter and will be applied in the remaining chapters.

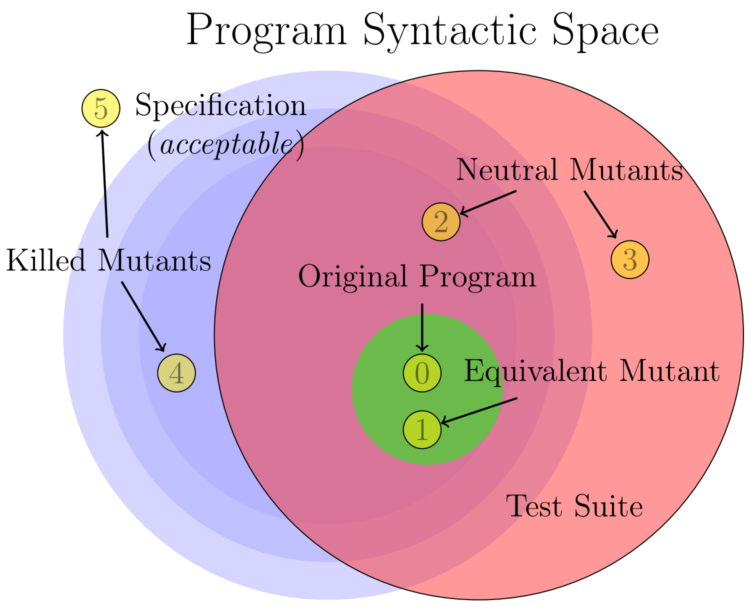


Figure 3.13: Syntactic Space of a Program. The set of programs satisfying the program specification are shaded blue (left), the set of programs passing the program's test suite are shaded red (right), and the set of equivalent programs are shown in green (center). Three classes of mutants are shown and labeled.

3.6.2 Re-interpretation of Mutation Testing

Many of the techniques presented in this chapter mirror those used in mutation testing [74]. However this dissertation makes a fundamentally different interpretation of the value and semantic stature of neutral mutants.

Figure 3.13 shows the syntactic space surrounding a program. This is similar to a fitness landscape; each point in the space represents a syntactically distinct program, and each program is associated with a semantic interpretation although that is not shown in the figure. Randomly mutating a program's syntactic representation can have several possible semantic effects, which are shown in the figure.

This highlights an alternative interpretation of our results: for every specification there exist multiple non-equivalent correct implementations. This emphasizes a different view of software from that of the mutation testing technique, namely, that for every program specification, all correct implementations are semantically equivalent.

To see how this follows from mutation testing, assume that $\exists$ programs a and b s.t. a is not equivalent to b ($a \neq b$) and both a and b satisfy specification S. Without loss of generality let a be the original program and b be a mutant of a. Let T be a test suite of S. According to Offut [123] there are two possibilities when T is applied to b. Either "the mutant is killable, but the test cases is insufficient" or "the mutant is functionally equivalent." The former case is impossible because b is assumed to be a correct implementation of S and thus should not be killed by any test suite of S. The later case is impossible because we assume $a \neq b$. By contraction, $\forall$ a and b satisfying the same specification S, $a = b$ or $\forall$ specification S $\exists! a$ s.t. a satisfies S.

Non-equivalent neutral mutants require a significant amount of developer attention. Each requires changes to the program test suite and possibly changes to the program specification and to the original program. The problem of differentiating between equivalent and non-equivalent mutants is termed the *equivalent mutant problem*, and is a significant problem in the practice of mutation testing (cf. *equivalent mutant problem* [74, Section II.C]). Such mutants are often not easily discriminated, taking an average of 15 minutes in one user study [58], and is only done correctly 80% of the time [2]. Beyond the problem of determining equivalence, it is not clear that adding tests to distinguish non-equivalent mutants is a useful way to drive test suite development [53].

Although these problems are well known, we were unable to find formal publications that experimentally identify the fraction of equivalent mutants (aside from work explicitly targeting Object-Oriented mutation operators which generate particularly high rates of equivalent mutants [146, 138, 122]).

Through our own review of the mutation testing literature we collected unreported counts of equivalent mutants from a number of papers [53, 121, 120, 40] that

Acronym	Description
`aar`	array reference for array reference replacement
`abs`	absolute value insertion
`acr`	array reference for constant replacement
`aor`	arithmetic operator replacement
`asr`	array reference for scalar variable replacement
`car`	constant for array reference replacement
`cnr`	comparable array name replacement
`crp`	constant replacement
`csr`	constant for scalar variable replacement
`der`	DO statement end replacement
`dsa`	DATA statement alterations
`glr`	GOTO label replacement
`lcr`	logical connector replacement
`ror`	relational operator replacement
`rsr`	RETURN statement replacement
`san`	statement analysis (replacement by TRAP)
`sar`	scalar variable for array reference replacement
`scr`	scalar for constant replacement
`sdl`	statement deletion
`src`	source constant replacement
`svr`	scalar variable replacement
`uoi`	unary operator insertion

Table 3.8: The Mothra mutation operators used in the landmark Mothra mutation testing system. Adapted from Table 1 of King *et al.* [83].

all used the Mothra [83] mutation operators and that found equivalent mutant rates of 9.92%, 6.75%, 6.24% and 6.17% respectively, indicating that equivalent mutants are common, but are less frequent than neutral variants.

The Mothra mutation operations are shown in Table 3.8. Mothra mutations are specific to the Fortran programming language are typically more specific than those used in this work. Mothra mutations include operations for constant, variable or array replacement which may be implemented using combinations of our *copy* and *insert*, and Mothra includes operators analogous to our own *delete* and *copy* (`sdl` and `uoi` in Table 3.8).

Of those categories of neutral variants described in Table 3.3, only categories 4, 5 and 6 could possibly have no impact on runtime behavior and could possibly be equivalent under Offut's strict definition of equivalence. Under the mutation testing paradigm, tests would be constructed to "kill" the remaining 29 of 35 neutral mutants. Given the sorting specification used (namely to print whitespace separated integer inputs in sorted order to `STDOUT` separated by whitespace), *none* of these classes of neutral mutations could be viewed as faulty. Consequently, any such extra tests constructed to distinguish them (e.g., for mutation testing) would *over-specify* the program specification. Rather than improving the test suite quality, such over-constrained tests could potentially judge future correct implementations as faulty.

In Chapter 4 we present an alternative to mutation testing which changes basic paradigm. Instead of immediately analyzing neutral variants manually, they are either deployed in an N-version scenario, or saved to aid in future debugging. This approach postpones the manual work of analyzing equivalent mutants (possibly forever), significantly reducing the burden on developer time. The alternative paradigm is discussed further in Section 4.3.

3.6.3 Legibility of Transformations

If automatically generated program transformations are to be incorporated into ongoing software development, they may need to be communicated back to software developers. This communication process is also important for manual review of evolved program adaptations. The legibility of program transformations differs for each level of representation. In general, transformations are less legible at the lower levels of program representation. Each representation is discussed in turn below.

CLang-AST Mutations at the CLang level provides the best legibility. Changes to the CLang AST may be automatically converted to source-level diffs which are properly indented and ready for either manual developer review or for direct application to program source.

CIL-AST Mutations at the AST level may be presented as differences in CIL processed source code, which are easily (if not automatically) translated into source level diffs. These changes can easily be applied to the original source code for integration into the software project moving forward.

LLVM and ASM Mutations at these levels are applied to either LLVM IR or compiled assembler code. Changes in program variants may easily be represented as assembler or IR diffs. Such diffs do allow for manual developer review, however, many software developers are not able or inclined to read such low-level languages, which are mainly written by compilers rather than by human developers.

Many compilers provide options to map specific assembler instructions to specific lines of code in the original program. In these cases it is easy to find the location of modifications in the source (e.g., using the `-c -g -Wa,-ahl=out.s` flags to `gcc`), however the content of the modification is often not easily, or not possibly, translated to the source code level, so there is currently no clear way to integrate changes at the LLVM or ASM levels into a software project for future use.

ELF Mutations in ELF representations which take place in the executable portion of the ELF file may be translated into changed assembler instructions using a disassembler such as `objdump` (part of the GNU Binutils collection[20]). In these cases ELF modifications are as legible as ASM modifications. In other cases ELF modifications will occur either in the data portion of an executable, or will not be easily disassembled, in which case manual review becomes a more difficult forensic exercise.

[20]`http://www.gnu.org/software/binutils`

3.6.4 Functional vs. Nonfunctional Evaluation

Nonfunctional evaluation leads to smoother fitness landscapes that are more amenable to search using evolutionary algorithms, as discussed in Section 2.1. The fitness evaluations described in this work that rely on the test suite of the original program return fitness values from a discrete set. This leads to a stepped fitness landscape composed of flat plateaus which provide no guidance to the evolutionary computation technique except at plateau boundaries. By contrast, the fitnesses returned by nonfunctional evaluation techniques (normally software profilers) are often continuous, and provide the gradients which evolutionary computation techniques are able to climb.

Function and nonfunctional properties of software are analogous to *discrete* and *continuous*, or *quantitative*, traits of biological organisms respectively. Discrete traits of biological organisms express phenotypes in a finite number of discrete classes and are often controlled by one or a few genes. Continuous traits are generally controlled by hundreds of genes, and are more common than discrete traits.

Chapter 7 presents an application that leverages the beneficial aspect of nonfunctional fitness evaluation. Section 8.2.2 posits possible techniques with which to address this limitation of current methods of functional fitness evaluation.

Chapter 4

Application: Program Diversity

Through the automated exploration of software neutral networks as in Section 3.4.1, large number of alternate implementations of programs, program variants, may be automatically generated. Such variants may be collected to form large populations of diverse implementations of a program. This process results in a novel form of artificial diversity.

Automated techniques of generating program variants with runtime or phenotypic diversity are collectively called *artificial diversity* [51]. These techniques make computer systems more secure against attack by making it harder to find, reproduce, and transfer exploits between machines. Such techniques typically involve randomizing some aspect of a computation; e.g., stack frame layouts [34], instruction set numbering [12], or address space layouts [147].

Unlike previous work in artificial diversity, neutral variants provide *implementation diversity* [33]. By randomizing implementation choices in the program rather than execution choices controlled by the operating system or environment, populations of neutral variants potentially offer increased security and may also potentially repair semantic defects present in the original program.

In this chapter we demonstrate that populations of diverse neutral variants can provide sufficient diversity to proactively repair latent defects in a program. In one possible use case, large populations of neutral variants could be retained by developers and used retroactively to quickly repair new bugs as they are discovered. When developers become aware of a new bug, the population could be checked quickly for variants that repair the bug.[1] If such variants exist they could be used to pinpoint the bug, and suggest a patch. This would be of practical benefit because developers are able to repair bugs more quickly with the help of such machine-generated diffs [159].

This chapter introduces an application of neutral networks in the automated generation of implementation diversity. The utility of neutral program variants to repair latent defects is accessed using held-out defects seeded into benchmark programs according to known fault distributions.

This work appeared in GPEM [144].

4.1 Methodology

We access the degree to which populations of neutral program variants may proactively repair bugs latent in an original program. We proceed by seeding latent bugs into real-world software, generating populations of variants of that software without knowledge of the seeded bugs, and then evaluating the degree to which software variants repair seeded bugs. The following steps were followed.

1. We manually seed each benchmark program in Table 4.1 with five bugs. These programs are selected to be characteristic of large real-world software projects.

[1]Although these variants are neutral to the original program with respect to the regression test suite, many neutral variants are computationally diverse from the original program (Section 3.3.2.3) and would *not* remain neutral under different test suites (in this case one which tests for the previously unknown bug).

The seeded bugs are drawn at random from an established defect distribution [55] and fault taxonomy [85], to ensure that our results generalize to the types of bugs found in real-world software.

2. For each defect, we manually write a test capable of detecting its presence in the program. These held out tests are withheld until step 4.

3. For each program, using the CIL-AST program representation we generate 5000 neutral variants using only the program regression test suite and *not* using the held out tests. These regression test suites are distributed with the programs, and are the test suites used by the software's developers themselves. As in Section 3.3.1.3 these test suites vary in quality and size. We apply mutations (defined in Section 3.1) uniformly at random and retain the resulting population of neutral variants.

4. Using the held out tests, we evaluate the populations of neutral variants noting how many variants pass each of the held out tests.

4.2 Results

Table 4.1 shows the results of this experiment. We find that in most (9 of 11) programs with five seeded bugs and 5000 neutral variants at least one neutral variant proactively repaired one of the bugs. When a bug repair was found in the neutral network we call this bug "repairable", multiple repairs were usually found with 17.25 proactive repairs found per repairable bug on average.

The types of bugs most commonly repaired were those that resemble the mutation operations. For example, we found multiple repairs for bugs that could be addressed by deleting problematic statements or clauses, or inserted clauses or statements to test for extra conditions. However, there was significant overlap between the types

Program	Total Bugs Fixed of 5	Bug Fixes	Fixes per Bug
`bzip`	2	63	31.5
`imagemagick`	2	8	4.0
`jansson`	2	40	20.0
`leukocyte`	1	1	1.0
`lighttpd`	1	73	73.0
`nullhttpd`	1	7	7.0
`oggenc`	0	0	0.0
`potion`	2	14	7.0
`redis`	0	0	0.0
`tiff`	0	0	0.0
`vyquon`	1	1	1.0
Average	1.09	18.82	17.3
Total	12	207	17.2

Table 4.1: Proactive repairs of seeded defects found in a population of 5000 neutral variants.

of bugs which were and were not repaired. We are not yet able to identify features which distinguish repairable and non-repairable bugs.

Through manual analysis of those variants that proactively repaired bugs, we found examples where the variant directly reverted the seeded bug by changing the same line of code in which the bug was seeded (3% of all repairs) or made a change within 5 lines of code on either side of the seeded bug (12% of all repairs). However, the majority of repairs (88%) were *compensatory* (compensatory mutations repair deleterious effects of changes in one gene by mutating a *different* gene, and are thought to be related to evolvability [131]) repairing the bug through changes elsewhere in the program.

These experiments included only five latent bugs per program. Most deployed programs have many more than five outstanding defects (e.g., 18,165 from October 2001 to August 2005 for Eclipse (V3.0) and 2,013 from May 2003 to August 2005 for Firefox (V1.0) [7]).

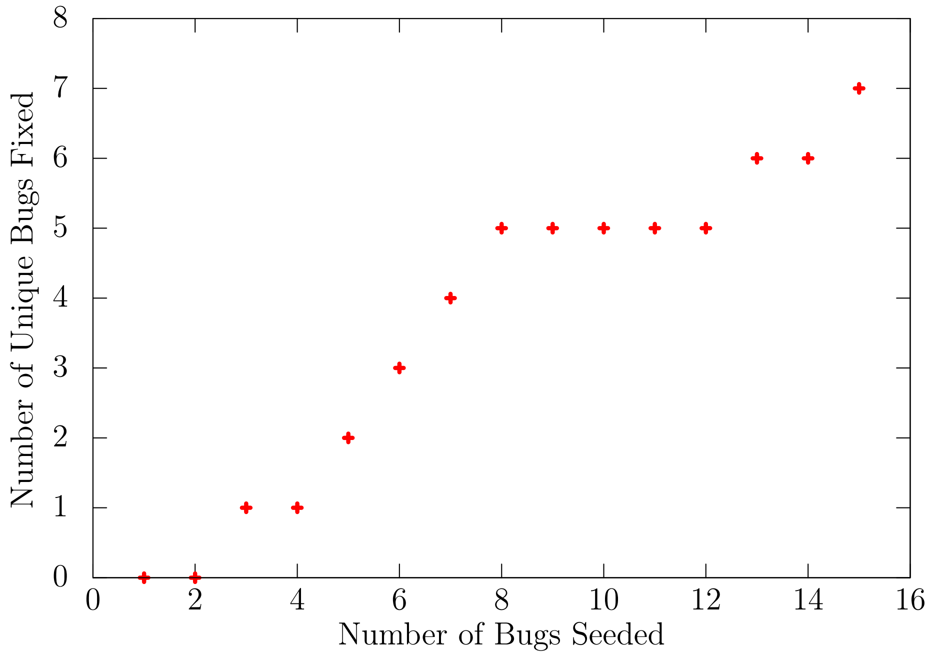


Figure 4.1: Number of proactive repairs in a population of 5000 neutral variants for the potion program as a function of the number of bugs seeded.

Figure 4.1 shows the number of distinct bugs repaired by 5000 neutral variants as a function of the number of defects seeded. The correlation between the number of proactive repairs found and the number of seeded bugs is 95%. If our results generalize and this correlation applied to the Eclipse and Firefox projects a population of 5000 neutral variants would repair 9000 and 1000 of the latent later-reported defects respectively, or in the case of Eclipse almost two bugs per neutral variant.

4.3 Discussion

This use of mutational robustness is analogous to software mutation testing, with the critical differences that (1) neutral mutants are retained rather than manually examined; (2) the test suite is not augmented to kill all mutants; and (3) the set of mutation operators considered is different. The commercial practice of mutation testing has been limited by the significant effort required to analyze mutants that pass the test suite. Such mutants must be manually classified, either as fully equivalent to the original program or non-equivalent, and the latter further classified as buggy or as superior to the original program (cf. *human oracle problem* [162]).

The methodology proposed in this section could provide an alternative to the traditional mutation testing practice, amortizing these labor-intensive steps by retaining a population of all such *neutral* variants. When a bug is encountered in the original program, it will be detected by running all variants against the bug and checking if some members of the population behave anomalously with respect to the result of the population. Then the non-failing variations need only then be analyzed to suggest a repair. This approach of deferring analysis until a potentially beneficial variation is found may be more feasible than traditional mutation testing, because it does not require exhaustive manual review of large numbers of program variants. Section 3.6.2 further discusses the relationship between this work and mutation testing.

Additional applications of the implementation diversity attained through neutral exploration could include running multiple neutral variants of a program simultaneously and automatically detecting differences in behavior (as in [71]), or deploying autonomous vehicles such as space vehicles with multiple neutral versions of critical software components providing fallback options in case of software failure. Techniques for reducing the size of neutral populations while maximizing the retained mutual diversity are explored in prior work [144].

Chapter 5

Application: Assembler- and Binary-Level Program Repair

Previous work on automated evolutionary program repair at the level of Cil ASTs demonstrated a wide range of applicability, for example, repairing 55 of 105 bugs in a large systematic study [95]. Lower level program representations such as the ASM and ELF levels (Section 3.1) offer a number of desirable features including generality to other languages than C, reduced source-code requirements (Section 3.1.3), faster expression times (Section 3.2.1) and greater coverage of the space of possible programs (Section 3.5).

Chapter 3 showed that the ASM and AST levels of representations have comparable mutational robustness. This chapter builds on that result to investigate both ASM and ELF level representations in terms of their ability to repair bugs (Section 5.3) and efficiency (Section 5.4). Performing program repair at these lower levels requires a new form of fault localization (Section 5.1), as the instrumentation-based method of AST fault localization [96] provides neither the granularity or efficiency necessary for these lower-level representations.

This work appeared in ASE 2010 [143] and ASPLOS 2013 [141].

5.1 Fault Localization

Fault localization is the process of determining the location of program faults [45]. The prior work with the AST program representation relied on fault localization to focus mutation operations to those portions of the program most likely related to the defect to be repaired. The fault localization process used synthesized traces of program execution on both good and bad input data to estimate the likelihood that each execution portion of the program was related to the fault [77].

These techniques required AST statement level program instrumentation. Analogous instrumentation solutions are problematic for lower level program representations, which may have no access to program source. Additionally. many existing code profilers (e.g., `gcov`) are source language specific making them unsuitable for application to ASM program representations.

We develop two methods of profiling, which are applicable to arbitrary assembler and ELF programs; a heavy-weight deterministic runtime harness and a lighter weight stochastic sampling technique. We compare the quality of results for the two techniques, finding them to be comparable, and conclude that the sampling technique is generally preferable. In both cases values of the program counter are collected which are easily converted into offsets in the vector program representation. Finally, stochastic samples are smoothed using a Gaussian convolution to improve their approximation of a full deterministic trace. The smoothing process is shown in Figure 5.1.

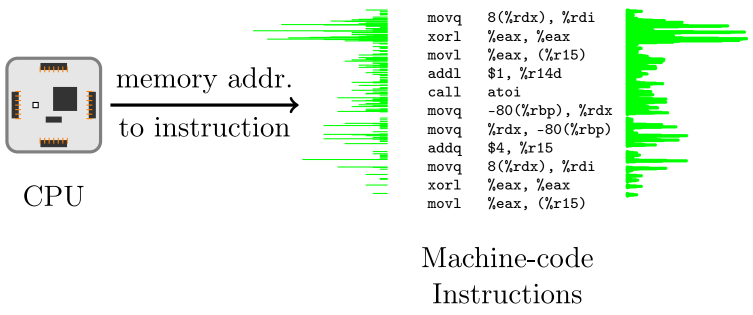


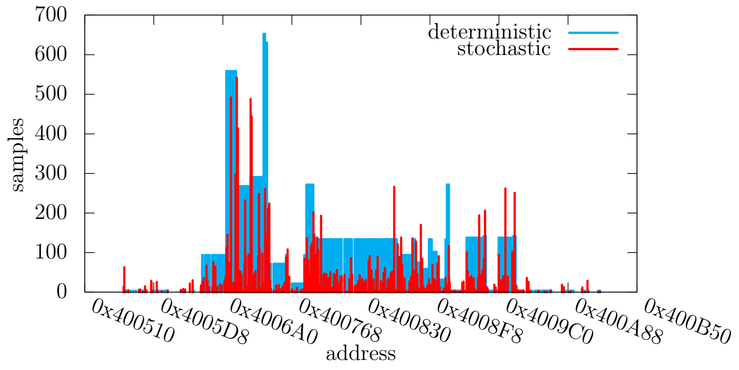
Figure 5.1: ASM and ELF level fault localization showing raw and smoothed samples from the merge-cpp benchmark shown in Table 5.1.

The deterministic sampling technique uses a simple `ptrace`-based runtime harness, which collects every value obtained by the program counter during execution.[1] The simplicity of this technique makes it preferable for short-running programs.

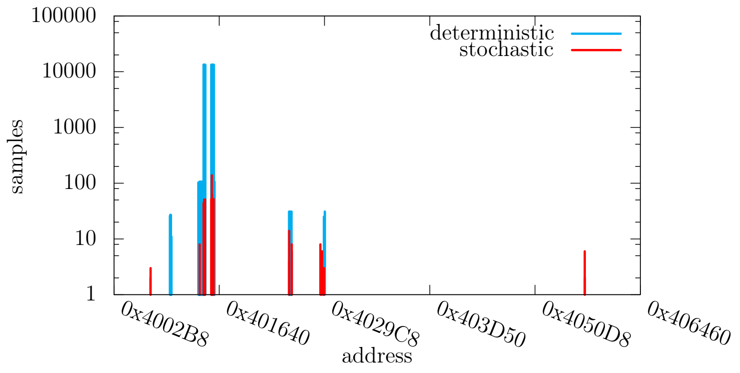
In most cases however, a lighter weight method is preferable. Stochastic sampling uses the `oprofile`[2] [100] system-wide profiler for Linux systems to sample the value of the program counter during execution (configuration settings including sampling frequency were left at their default values). Oprofile returns a count of the total number of times each instruction in the program was sampled. Sampling only approximates control flow and is vulnerable to gaps and over-sampling of certain instructions (e.g., those inside of loops), in addition, the fine granularity of samples often under-estimates the total number of executed instructions by sampling single instructions in long runs of contiguous executed instructions.

To compensate for these issues, we apply a 1-D Gaussian convolution (Equation 5.1) to the sampled addresses with a radius of 3 assembler instructions. The resulting smoothed address of each instruction x is then a weighted sum $G(x)$ of its raw sample count and it's 3 neighbors $(x + i)$ on either side.

[1]`https://github.com/eschulte/tracer`
[2]`http://oprofile.sourceforge.net/news`



(a) `merge`



(b) `deroff`

Figure 5.2: Comparison of stochastic sampling (shown in red) and deterministic fault localization (shown in blue).

$$G(x) = \sum_{i=-3}^{3} F(x+i) \times \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}i^2} \tag{5.1}$$

Gaussian convolution is commonly used to smooth data in fields such as computer vision [148]. However to our knowledge it had not previously been used for fault localization. Figure 5.2 compares stochastic and deterministic fault localization for two programs, `deroff` and merge sort. In both cases the light weight stochastic samples have a high fidelity to the full deterministic samples. Merge sort provides an example of a program with high fault localization coverage, and `deroff` provides an example of a program with low fault localization coverage.

	Program Size				
	C	ASM	ELF		
Program	LOC	LOC	Bytes	Program Description	Defect
`atris`	9578	39153	131756	graphical tetris game	buffer exploit
`ccrypt`	4249	15261	18716	encryption utility	segfault
`deroff`	1467	6330	17692	document processing	segfault
`flex`	8779	37119	73452	lexical analyzer generator	segfault
`indent`	5952	15462	49384	source code processing	infinite loop
`look-s`	205	516	1628	dictionary lookup	infinite loop
`look-u`	205	541	1784	dictionary lookup	infinite loop
`merge`	72	219	1384	merge sort	improper sorting
`merge-cpp`	71	421	1540	merge sort (in C++)	improper sorting
`s3`	594	767	1804	sendmail utility	buffer overflow
`uniq`	143	421	1288	duplicate text processing	segfault
`units`	496	1364	3196	metric conversion	segfault
`zune`	51	108	664	embedded media player	infinite loop
total	31862	117682	304288		

Table 5.1: Benchmarks used in program repair experiments using the ASM and ELF level program representations. Each program includes one bug described in the "Defect" column. The "Program Size" columns give the size of the programs in lines of code "LOC" for AST and ASM representations and in bytes of program data for the ELF representation.

5.2 Benchmarks

To evaluate the effectiveness of repair at the ASM and ELF levels, a number of benchmark programs used in previous work at the CIL-AST level were selected. The success rates and search metrics were collected and compared to earlier work.

The suite of benchmark programs is shown in Table 5.1 together with the size at different representation levels, a brief program description, and a defect description. The selected programs cover a wide range of both bugs and security vulnerabilities. All programs except for the C++ version of merge sort are taken directly from previous on program repair at the CIL-AST level [161].

	% Success			Expected Fitness Evaluations		
Program	AST	ASM	ELF	AST	ASM	ELF
`atris`	83	0	5	27.44	†	48806.00
`ccrypt`	100	100	100	7.00	673.00	25.00
`deroff`	100	98	100	48.00	50.00	454.00
`flex`	6	1	0	78340.50	496255.00	†
`indent`	4	41	0	62737.25	13517.48	†
`look-s`	100	100	100	41.00	71.00	3.00
`look-u`	100	100	100	90.00	16.00	19.00
`merge`	54	100	84	4456.85	621.00	1008.19
`merge-cpp`		100	79	†	314.00	2135.2658
`s3`	100	96	50	4.00	4.00	95.00
`uniq`	100	100	100	8.00	46.00	8.00
`units`	91	13	51	930.23	57374.63	8538.47
`zune`	100	100	100	17.00	26.00	45.00
average	78.17	70.75	65.83	622.45	6542.40	1132.85
w/o `units`	77.00	76.00	67.18	583.98	188.38	207.15

Table 5.2: Evaluation of the effectiveness of ASM and ELF level representations. "% Success" gives the percentage of random seeds for which a valid repair is found within 5000 runs of the full test suite. "Expected Fitness Evaluations" counts the expected number of evaluations per repair (Equation 5.2). † Indicates that there were no successful repairs in 5000 fitness evaluations. Rows with † were excluded when calculating average "Expected Fitness Evaluations".

5.3 Effectiveness

Table 5.2 compares the ability of the ASM and ELF level representations to repair defects that were also repaired in previous work at the Cil-AST level (Weimer *et al.* [161]). Due to the smaller scale of the program transformations performed at ELF and ASM levels (Section 3.1.2), we expected the repair process at these levels to be both slower and less successful. We were surprised to find comparable overall success rates between the ASM and ELF level repairs (70.75% and 65.83% respectively) and the AST level 78.18%. Using the Fisher's Exact test to compare success rates between Cil-AST and the lower levels we find no significant difference, with p-values of 1 between AST and ASM, and 0.294 between AST and ELF.

The “Expected Fitness Evaluations” column reports the expected number of fitness evaluations per repair (including failed repair attempts). The calculation of expected fitness evaluations is given in Equation 5.2.

$$\begin{aligned} expected &= fit_s + (run_s - 1) \times fit_f \text{ where} \\ fit_s &= \text{average evaluations per successful run} \\ fit_f &= \text{average evaluations per failed run} \\ run_s &= \text{average runs per success} \end{aligned} \tag{5.2}$$

More surprising was the reduced number of fitness evaluations required to find a repair at these lower levels shown in the “w/o units” average (which removes the outlier “`units`” program) of the expected fitness evaluations for each representation. The ASM and ELF level repairs required 188.38 and 207.15 fitness evaluations on average respectively as compared to 583.98 fitness evaluations on average for CIL-AST level repairs. These results suggest that even though mutations at these lower levels affect smaller portions of the program, the repairs at these levels are located more closely (in terms of mutation) or more densely around the original program. To provide intuition for this statement, consider the repair for the simple merge sort bug which incorrectly sorts some inputs. A valid repair is to swap the `then` and `else` branches of the following `if` statement.

```
if(left[l-mid-1]<=right[0]) { /* fix: swap branches */
  result=list;
} else {
   result=merge(left,l-mid, right,mid);
}
```

At the Cil-AST level this repair is only accomplished by 2 of 4900 possible swap mutation operations.[3] At the lower levels this repair is accomplished by simply deleting the `cmpl` instruction in the following assembler code.

```
cmp %eax, %edx ;; fix: delete instruction
jg   .L12
mov -72(%rbp), %rax
```

This is 1 of only 280 possible delete mutation operations,[4] and is much more easily found.

5.4 Efficiency

Having found comparable effectiveness across all levels, we next consider the efficiency of the repair process by level of representation. This includes both static properties such as the size of the installed toolchain required to perform repair (Section 3.2.1), as well as runtime properties such as the time required to perform repair. In general the ASM and ELF representations perform automated program repair more efficiently than the higher Cil-AST level as shown in Table 5.3.

Runtime The runtime of the automated repair process is dominated by the time taken to perform fitness evaluations (Section 3.2). The time taken for a fitness evaluation includes both the time required to run the test suite, and the time required to express the program as an executable (i.e., compile and link), the lower level representations are expressed much more efficiently because they don't need to be compiled (ASM) or compiled and linked (ELF) (Section 3.2.1). This effect is compounded by the fact that fewer expected total fitness evaluations were required at the lower levels as shown in the previous section. As

[3]Merge sort has 70 total Cil-AST statements. Each swap selects two statements for a total of $70 \times 70 = 4900$ possible swap operations.

[4]Merge sort has 280 total ASM instructions. Each deletion selects one instruction for a total of 280 possible deletion operations.

	Memory (MB)			Runtime (s)		
Program	AST	ASM	ELF	AST	ASM	ELF
`atris`	2384*	2384*	496	22.87	†	385.63
`ccrypt`	6437*	3338*	334	39.15	342.23	21.58
`deroff`	1907*	811	453	37.33	1366.61	292.88
`flex`	691	381	162	1948.84	1125.44	†
`indent`	3242*	1669*	572	3301.88	3852.47	†
`look-s`	420	62	29	747.59	353.81	6.00
`look-u`	430	52	62	12.68	6.38	3.66
`merge`	152	45	57	842.74	100.93	161.35
`merge-cpp`	†	50	60	†	121.87	90.56
`s3`	152	76	43	14.43	23.46	28.02
`uniq`	358	72	72	105.18	3.46	7.18
`units`	572	162	95	1075.16	18778.70	501.54
`zune`	76	17	29	36.93	28.79	71.49
average	1401.75	755.75	200.33	323.47	2333.82	121.52
w/o `units`	1242	559	135	229.50	278.20	74.02

Table 5.3: Evaluation of defect repair efficiency at the ASM and ELF level representations. "Memory" reports the average max memory required for a repair (as reported by the Unix `top` utility). "Runtime" reports the average time per successful repair in seconds. † Indicates that there were no successful repairs in 5000 fitness evaluations. Rows with † are excluded when calculating "Memory," "Runtime," and "Expected Fitness Evaluations" averages.

a result the runtime for repair is significantly reduced for the lower levels of program representation.

Despite the reduced number of fitness evaluations and reduced expression time, ASM level repairs take *longer* on average than AST level repairs. This might be due to an increased likelihood of transformations at the ASM resulting in

	ASM	ELF
Runtime	-21.2%	67.7%
Disk	47.4%	95.2%
Memory	55.0%	89.1%

Table 5.4: Decrease in resource requirements at the ASM and ELF level representations compared to the `{\sc Cil}`-AST level representation. "Runtime" and "Memory" numbers exclude `units` as an outlier.

new bugs, such as infinite loops, which increase the running time of the test suite.

Disk usage The disk usage during repairs differs dramatically between levels of representation, as expected. The reduced disk footprint of the ASM and ELF levels is primarily due to the fact that these lower-level representations do not need a compiler, or a compiler and linker respectively, to express programs as executables.

Memory The working memory of the repair process is dominated by the space required to hold the population of candidate repairs variants. This space is dictated by the size of individual program representations in memory. The ASM and ELF level program representations are more space-efficient. Instead of storing a tree of source-code statements as in the AST level, the lower levels store a vector of text assembly instructions at the ASM level and a vectors of byte-sequences at the ELF level.

5.5 Discussion

This chapter described experiments of the ability of the ASM and ELF level program representations to repair bugs taken from previous program repair work at the CIL-AST level. To efficiently perform program repair over these lower level representations changes to the program repair process were required, including a lighter weight stochastic method of fault localization (Section 5.1).

Program repair at these lower levels was found to be as effective as program repair at the CIL-AST level (Section 5.3) and more efficient in terms of memory consumption, disk footprint and runtime (Section 5.4).

Chapter 6

Application: Patching Closed Source Executables

The ability to manipulate ELF files directly obviates the need for *any* access to software development resources such as source code or build toolchains. This suggests the possibility of modifying proprietary closed-source applications. An interesting use case, for example, is repairing security vulnerabilities in closed-source executables. However, this use case raises new challenges, such as how to conduct automated program repair without access to a regression test suite. We demonstrate this application by repairing multiple security vulnerabilities in the NETGEAR WNDR3700 wireless router.[1]

Router bugs are a significant issue, ranging from the bug in CISCO's IOS, which on February 16th 2009 caused outages in nearly every country worldwide [167], to security vulnerabilities in home routers like NEGEAR [35] or the recent D-Link bug [49]. Security bugs are particularly problematic, especially because major software vendors commonly delay releasing patches to security exploits. In a study of

[1]`http://www.netgear.com/home/products/networking/wifi-routers/wndr3700.aspx`

high- and medium-risk vulnerabilities in Microsoft and Apple products between 2002 and 2008, for example, about 10% of vulnerabilities were found not to be patched within 150 days of disclosure, and on any given date about 10 vulnerabilities and over 20 vulnerabilities were public and un-patched for Microsoft and Apple respectively [54].

In recent years, a variety of automated methods for program repair have successfully repaired defects in real software (see Section 2.4). The ELF program representation introduced in Chapter 3 allows for the transformation and evaluation of binary executables without the need for any access to developer resources such as source code, or built toolchains. In Chapter 5 we described automated repair methods based on evolutionary computation to repair defects directly in x86 and ARM ELF files. Chapter 5, however, relies on a regression test suite to define the required functionality of the program under repair. Here we consider a setting in which neither source code nor test suites are available, and there is no special information or cooperation from the vendor.

Software vendors are often slow to respond to security vulnerabilities after exploits have been discovered. This leads to a dire situation for end users who lack product source code and must wait for a patch to be released by the vendor. Rather than waiting for vendor-delivered patches, we present an alternative approach in which newly discovered exploits drive an automated repair technique capable of patching vulnerabilities, even without access to source code or special information from the software vendor. A user-produced patch could be installed temporarily for internal protection, redistributed with the exploit (reporting an exploit with a patch in hand has been shown to reduce the total number of attacks [8]), or sent to the software vendor to reduce development time for the official patch [159].

This Chapter describes how the Genprog repair method can be applied to this new use case. Extensions to earlier work include: Repairing security vulnerabilities in router binaries; special processing to handle stripped ELF files; operating without fault localization information; operating without a pre-existing regression test suite to define required program behavior; and the discovery of multiple iterative repairs in a binary.

We demonstrate the method by repairing two recently discovered exploits in version 4 of NETGEAR's WNDR3700 wireless router *before* NETGEAR released patches publicly for the exploits (at the time of writing NETGEAR has not publicly addressed the exploits). Without the use of any regression tests to guide the search, we find that 80% of the automatically generated repairs for the example vulnerabilities retain program functionality. When user-created tests of required functionality are incorporated in an interactive process, success quickly increases to 100% of the proposed repairs.

To encourage reproducibility and to allow others to patch future vulnerabilities, we provide a companion source repository[2] for this Chapter. It contains the instructions, source code, and tooling needed to extract, execute and repair the binary NETGEAR router image vulnerabilities, and to regenerate the analyses, tables, and figures included in this chapter.

The remainder of the chapter reviews two recent exploits of NETGEAR WNDR3700 (Section 6.1); demonstrates the feasibility of running the NETGEAR firmware in a VM sandbox (Section 6.2.1); describes the automated program repair technique (Sections 6.2.2 and 6.2.3); evaluates effectiveness and quality of repairs (Section 6.3); and discusses implications and limitations (Section 6.4).

This work is previously unpublished.

[2]`https://github.com/eschulte/netgear-repair`

6.1 Description of Exploits

We describe two current exploits in version 4 of the NETGEAR WNDR3700 wireless router. The popularity of this router implies that vulnerable systems are currently widespread. For example, the "shodan"[3] device search engine returned hundreds of vulnerable publicly accessible WNDR3700 routers at the time of writing. Both exploits exist in the router's internal web server in a binary executable named `net-cgi`, and both are related to how `net-cgi` handles authentication [35].

The vendor-deployed binary is insecure in at least two ways:

1. Any URL starting with the string "BRS" bypasses authentication.
2. Any URL including the substring "`unauth.cgi`" or "`securityquestions.cgi`" bypasses authentication. This applies even to requests of the form `http://router/page.html?foo=unauth.cgi`, meaning that the vulnerability effectively applies to all internal webpages.

Many administrative pages start with the "`BRS`" string, providing attackers with access to personal information such as users passwords, and by accessing the page `http://router/BRS_02_genieHelp.html` attackers can disable authentication completely and permanently across reboots.

6.2 Automated Repair Method

Our repair technique for this vulnerability consists of three stages:

1. Extract the binary executable from the firmware and reproduce the exploit (Section 6.2.1).

[3]`http://www.shodanhq.com/search?q=wndr3700v4+http`

2. Use evolutionary techniques to search for repairs by applying random mutations (and crossover) to the stripped (without symbols or section tables) MIPS ELF binary (Section 6.2.2).

3. Construct test cases lazily, as needed, to improve the quality of unsatisfactory candidate repairs (Section 6.2.3).

The first step in repairing the `net-cgi` executable is to extract it and the router file system from the firmware image distributed by NETGEAR. Using the extracted file system and executable we construct a test harness that can exercise the vulnerabilities in `net-cgi`. This test harness is used by the repair algorithm to evaluate candidate repairs and to identify when repairs for the vulnerabilities have been found.

6.2.1 Firmware Extraction and Virtualization

NETGEAR distributes firmware with a full system image for the WNDR3700 router, which includes the router file system that has the vulnerable `net-cgi` executable. We extracted the file system using the `binwalk`[4] firmware extraction tool, which scans the binary data in the raw monolithic firmware file, searching for signatures identifying embedded data sections, including `squashfs` [103] that hold the router's file system.

The router runs on a big-endian MIPS architecture, requiring emulation on most desktop systems to safely reproduce the exploit and evaluate candidate repairs. We used the QEMU system emulator [15] to emulate the MIPS architecture in a lightweight manner with Debian Linux also run in emulation. The extracted router file system is copied into the emulated MIPS Linux system. A number of special directories (e.g., `/proc/`, `/dev/`) are mounted inside the extracted file system and bound to the corresponding directories on the virtual machine. At this point, com-

[4]`http://binwalk.org`

mands can be executed in an environment that closely approximates the execution environment of the NETGEAR router by using the `chroot` command to confine executable access to within the extracted NETGEAR file system. Additional minor adjustments are described in the reproduction documentation.[5]

At this point the NETGEAR router can be run under virtualization. In particular, the router's web interface can be accessed either using an external web browser or the `net-cgi` executable can be called directly from the command line.

6.2.2 Automated Program Repair and ELF Files

The repair algorithm constructs a population of 512 program variants, each with one or more random mutations (Chapter 3). This population is evolved through an iterated process of evaluation, selection, mutation, and crossover until a version of the original program is found that repairs the bug. "Repair" in this context is defined to mean that it avoids the buggy behavior and does not break required functionality. In Chapter 5, execution traces were collected during program execution and used as a form of fault localization to bias random mutations towards the parts of the program most likely to contain the bug. Our decision not to use fault localization is explained in Section 6.3.2.2.

The basic Genprog repair algorithm was modified in several ways to address the unique scenario of a user repairing a faulty binary executable, without access to a regression test suite (Section 6.2.3), and without the fault localization optimization.

6.2.2.1 Challenge: Mutating Stripped Binaries

Executable programs for Unix and embedded system are commonly distributed as ELF [32] files. Each ELF file contains a number of headers and tables containing

[5]`http://eschulte.github.io/netgear-repair/INSTRUCTIONS.html`

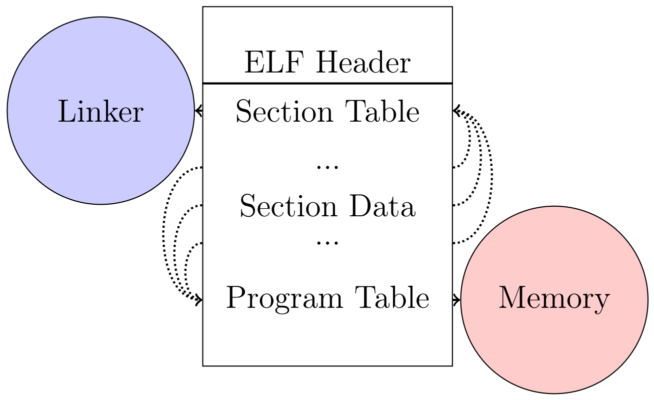


Figure 6.1: Sections and their uses in an ELF file.

administrative data, and sections holding program code and data. The three main administrative elements of an ELF file are the ELF header, the section table and the program table (see Figure 6.1). The ELF header points to the section table and the program table, the section table holds information on the layout of sections in the ELF file on disk, and the program table holds information on how to copy sections from disk into memory for program execution.

Although the majority of ELF files include all three of the elements shown in Figure 6.1, only the ELF Header is guaranteed to exist in all cases. In executable ELF files, the program table is also required, and similarly, in linkable files the section table is required.

We extend Chapter 5, which repaired unstripped Intel and ARM files [141]. In that work the `.text` section of the ELF file was modified by the mutation and crossover operations, but in this case `net-cgi` does not include key information on which the earlier work relied, namely the section table and section name string table. This information was used to locate the `.text` section of the ELF file where program code is normally stored. The data in the `.text` section were then coerced into a vector of assembly instructions (the *genome*) on which the mutation operations were defined. Our extension removes this dependence by concatenating the data of

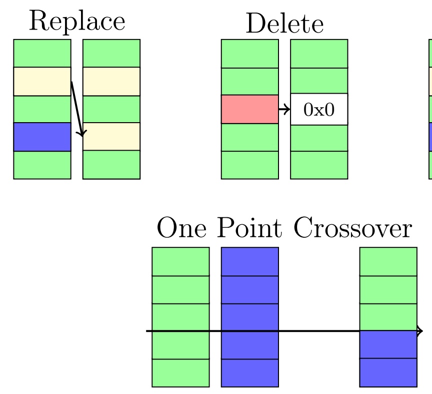


Figure 6.2: Mutation and Crossover operations for stripped MIPS ELF files. The program data are represented as a fixed length array of single-word sections. These operators change these sections maintaining length and offset in the array.

every section in the program table that has a “loadable” type to produce the genome. These are the sections whose data are loaded into memory during program execution.

Mutation operations must change program data without corrupting the structure of the file or breaking the many addresses hard coded into the program data itself (e.g., as destinations for conditional jumps). In general, it is impossible to distinguish between an integer literal and an address in program data, so the mutation operations are designed to preserve operand absolute sizes and offsets within the ELF program data. This requirement is easily met because every argumented assembly instruction in the MIPS RISC architecture is one word long [61]. “Single point crossover” is used to recombine two ELF files. An offset in the program data is selected, then bytes from one file are taken up to that offset and bytes from the other file taken after that offset. This form of crossover works especially well because all ELF files will have similar total length and offsets. The mutation and crossover operations used to modify stripped MIPS ELF files are shown in Figure 6.2.

6.2.3 On-Demand Regression Testing

Our approach to program repair relies on the ability to assess the validity of program variants. The mutations are random in the sense that they do not take into account or preserve the semantics of the program. They are more likely to create new bugs or vulnerabilities than they are to repair undesired behavior, and the method requires an evaluation scheme to distinguish between these cases.

Instead of relying on a pre-existing regression test suite, we assume only that a demonstration of the exploit provides a single available failing test. By mutating programs without the safety net of a regression test suite, the evolved “repairs” often introduce significant regressions. However, by applying a strict minimization process after the primary repair is identified, these regressions are usually removed (as in the Genprog technique described in Section 2.4). The minimization reduces the difference between the evolved repair and the original program to as few edits as possible using delta debugging[6]. The interactive phase of the repair algorithm asks the user to identify any regressions that remain after the delta debugging step. In the case of the NETGEAR router bug, repaired versions of the `net-cgi` executable web-server are run in simulation and a user uses a web browser to manually use the web server. High-level pseudocode for the repair algorithm is show in Figure 6.3.

Our method is thus an interactive repair process in which the algorithm searches for a patch that passes every available test (starting with only the exploit), and then minimizes it using delta debugging. In a third step, the user evaluates its suitability. If the repair is accepted, the process terminates. Otherwise, the user supplies a new regression test that the repair fails (a witness to its unsuitability) and the process repeats. In Section 6.3 we find that 80% of our attempts to repair the NETGEAR WNDR3700 exploits did not require any user-written regression tests.

[6]`https://github.com/eschulte/delta-debug`

```
Input:  Vulnerable Program, original : $ELF$
Input:  Exploit Tests, exploits : $[ELF \rightarrow Fitness]$
Input:  Interactive Check, goodEnough : $ELF \rightarrow [ELF \rightarrow Fitness]$
Output:  Patched version of Program
 1: let $new$ ← null
 2: let $fitness$ ← null
 3: let $suite$ ← exploits
 4: repeat
 5:     let full ← evolSubroutine(original, suite)
 6:     $new$ ← minimize()
 7:     let $newRegressionTests$ ← goodEnough(new)
 8:     suite ← suite + +newRegressionTests
 9: until  $length$(newRegressionTests)$(-1cm : 1.5cm)circle(1cm)0$
10: return  new
```

Figure 6.3: High-level Pseudocode for interactive lazy-regression-testing repair algorithm.

The `evolSubroutine` in Figure 6.3 is organized similarly to previous work [96], but it uses a *steady state* evolutionary algorithm (Section 2.2.2). Figure 6.4 gives the high-level pseudocode.

Note that every time the user rejects the solution returned by `evolSubroutine`, the evolved and minimized solution is discarded and a new population is generated by recopying the original in `evolSubroutine`.

6.3 Repairing the NETGEAR Exploits

We first describe the experimental setup used to test the repair technique on the NETGEAR WNDR3700 vulnerability (Section 6.3.1). We then analyze the results of ten repair attempts (Section 6.3.2).

```
Input: Vulnerable Program, original : $ELF$
Input: Test Suite, suite : $[ELF \rightarrow Fitness]$
Parameters: $populationSize$, $tournamentSize$, $crossRate$
Output: Patched version of Program
 1: let $fitness \leftarrow$ evaluate(original, suite)
 2: let $pop \leftarrow$ populationSize copies of $\langle$original, fitness$\rangle$
 3: repeat
 4:     if Random() $<$ $CrossRate$ then
 5:         let $p_1 \leftarrow$ crossover(tournament(pop, tounamentSize, $+$))
 6:         let $p_2 \leftarrow$ crossover(tournament(pop, tounamentSize, $+$))
 7:         let $p \leftarrow$ crossover($p_1$, $p_2$)
 8:     else
 9:         $p \leftarrow$ tournament(pop, tounamentSize, $+$)
10:     end if
11:     let $p' \leftarrow$ Mutate($p$)
12:     let $fitness \leftarrow$ evaluate(suite, $p'$)
13:     incorporate($pop$, $\langle p'$, Fitness(Run($p'$))$\rangle$)
14:     if length(pop) $>$ maxPopulationSize then
15:         evict(pop, tournament(pop, tounamentSize, $-$))
16:     end if
17: until fitness $>$ length(suite)
18: return $p'$
```

Figure 6.4: High-level Pseudocode for the steady state parallel evolutionary repair subroutine.

6.3.1 Methodology

All repairs were performed on a server-class machine with 32 physical Intel Xeon 2.60GHz cores, Hyper-Threading and 120 GB of Memory. We used a test harness to assess the fitness of each program variant (Section 3.2) and report parameters used in the experiments (Section 6.3.1.2). An overview of the experimental configuration is provided in Figure 6.5.

6.3.1.1 Fitness Evaluation

We used 32 QEMU virtual machines, each running Debian Linux with the NETGEAR router firmware environment available inside of a `chroot`. The repair algo-

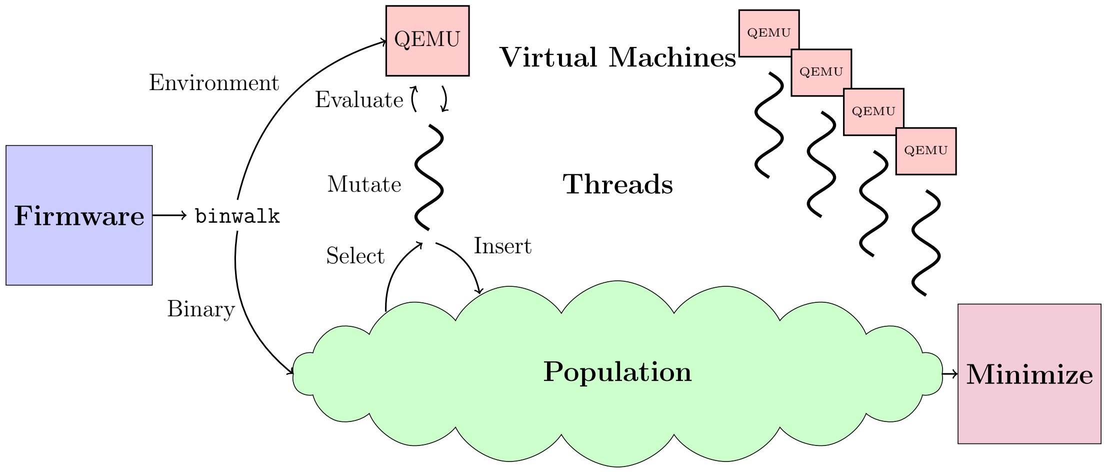


Figure 6.5: A high level view of the tooling used to repair a closed source router bug.

rithm uses 32 threads for parallel fitness evaluation. Each thread is paired with a single QEMU VM on which it tests fitness.

The test framework includes both a host and a guest test script. The host script runs on the server performing repair and the guest script runs in a MIPS virtual machine. The host script copies a variant of the `net-cgi` executable to the guest VM where the guest test script executes `net-cgi` the command line and reports a result of `Pass`, `Fail`, or `Error` for each test. These values are then used to calculate the variant's scalar fitness as shown in Equation 6.1.

$$fitness = \sum_{t \in \text{Test Suite}} \begin{cases} 2 & \text{if } \texttt{PASS} \\ 1 & \text{if } \texttt{FAIL} \\ 0 & \text{if } \texttt{ERROR} \end{cases} \tag{6.1}$$

`Pass` indicates that the program completed successfully and produced the correct result, `Fail` indicates that the program completed successfully but produced an incorrect result, and `Error` indicates that the program execution did not complete successfully due to early termination (e.g., because of a segfault) or by a non-zero `ERRNO` exit value.

6.3.1.2 Repair Parameters

Repair uses the following parameters. The maximum population size is 2^9 individuals, selection is performed using a tournament size of two.[7] When the population overflows the maximum population size, an individual is selected for eviction using a "negative" tournament in which the lowest fitness individual is selected for eviction. Newly generated individuals undergo crossover two-thirds of the time.

These parameters differ significantly from those used in previous evolutionary repair algorithms (e.g., [50, 95, 97]). Specifically, we use larger populations (512 instead of 40 individuals), running for many more fitness evaluations ($\leq$100,000 instead of $\leq$400). However, the parameters used here are in line with those used in other evolutionary program repair publications given the size of the `net-cgi` binary, and they help compensate for the lack of fault localization information.

The increased memory required by the larger population size is offset by the use of a steady state evolutionary algorithm (Section 2.2.2), and the increased computational demand of the greater number of fitness evaluations is offset by parallelization of fitness evaluation.

[7]When the fitness of all variants in the population has been evaluated, the fitness values are used to select one individual for subsequent modifications in the next generation. We use *tournament selection* where each tournament chooses a subset of two (the tournament size) randomly from the population and the individual with higher fitness wins the tournament and is copied into the population.

6.3.2 Experimental Results

We report results for the time typically taken to generate a repair (Section 6.3.2.1), the effect of eliminating fault localization (Section 6.3.2.2), and the impact of the minimization process (Section 6.3.2.3), both with respect to the size of the repair in terms of byte difference from the original and in terms of the fitness improvement.

6.3.2.1 Repair Runtime

In 8 of the 10 runs of the algorithm (with random restarts), the three exploit tests alone were sufficient to generate a satisfactory repair (determined using a withheld regression test suite hand-written by the authors[8]), and the third phase of user-generated tests was not required.

In these cases the repair process took an average of ~36,000 total fitness evaluations requiring on average 86.6 minutes to find a repair using 32 virtual machines for parallelized fitness evaluation.

6.3.2.2 Repair without Fault Localization

In the NETGEAR scenario we do not use any form of fault localization. While this might reduce the efficiency of repair, it is a benefit in cases where the use of fault localization would over-constrain the search operators. The limitation, as in Genprog, of program transformations to *only* those portions of a program exercised by the failing program inputs prevents valid repairs from being found if they require modification to statements outside of those executed by the failing input (in e.g., type definitions, global variables, or data sections in an ELF file).

[8]`https://github.com/eschulte/netgear-repair/blob/master/bin/test-cgi`

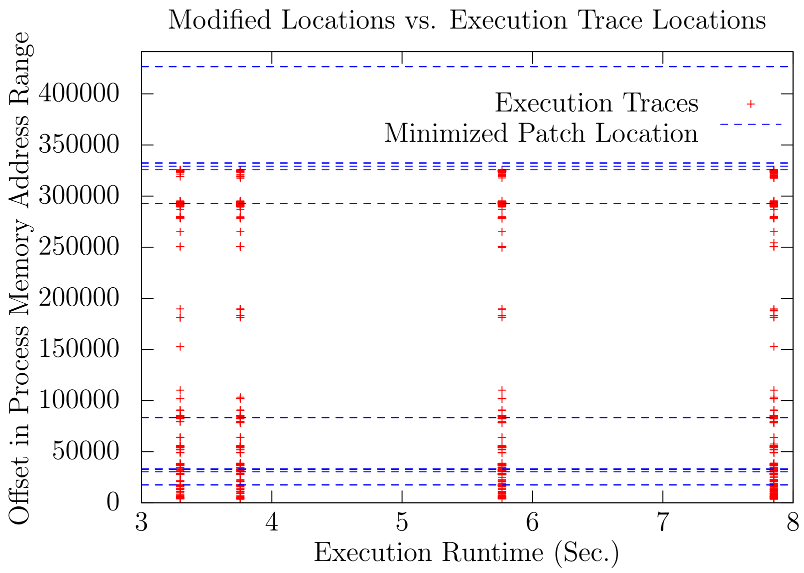


Figure 6.6: Fixes occur in different locations from execution traces: The location of every edit in a minimized successful repair is plotted as a horizontal line. Each vertical column shows points of execution traces from one test suite. Test suites shown from left to right are 3 tests (exploit tests only), 4, 7, and 11 tests (exploit and author-generated regression tests), with 330, 399, 518, and 596 sampled execution locations respectively. Code modifications occur in different locations from execution traces

One of the NETGEAR exploits exemplifies this issue. As shown in Figure 6.6, fault localization might have prevented the repair process from succeeding. The figure shows that many of the program edit locations for successful repairs were not visited by the execution trace. In fact, only 2 of the 22 program locations modified by successful repairs were within 3 instructions of the execution traces. Although surprising, this result suggests that earlier work, which confines edit operations to execution traces, would likely be unable to repair the NETGEAR bugs.

6.3.2.3 The impact of Minimization

In some cases the initial suggested repair, known as the *primary* repair, was not satisfactory. For example, suggested repairs sometimes worked when `net-cgi` was called directly on the command line but not through the embedded uHTTPd webserver,[9] or the repaired file failed to serve pages not used in the exploit test. However, Table 6.1 shows that in most cases the minimized version of the repair was satisfactory, successfully passing all hand-written regression tests, even those not used during the repair process.

Run	Fit Evals	Full Diff	Min Diff	Full Fit	Min Fit
0	90405	500	2	8	22
1	17231	134	3	22	22
2	26879	205	2	21	22
3	23764	199	2	19	22
4	47906	319	2	6	6
5	13102	95	2	16	22
6	76960	556	3	17	22
7	11831	79	3	20	22
8	2846	10	1	14	14
9	25600	182	2	21	22
mean	33652.4	227.9	2.2	16.4	19.6

Table 6.1: The evolved repair before and after minimization. In these columns "Full" refers to evolved solutions before minimization and "Min" refers to solutions after. Columns labeled "Diff" report the number of diff hunks against the original program data. The columns labeled "Fit" report fitness as measured with a full regression test suite, including the exploit tests. The maximum possible fitness score is 22 (using the fitness function in Equation 6.1 with all 11 tests), indicating a successful repair.

As shown in Table 6.1, the initial evolved repair differed from the original at over 200 locations[10] on average in the ELF program data, while the minimized repairs differed at only 1–3 locations on average. This great discrepancy is due to the accumulation of candidate edits in non-tested portions of the program data. Since these portions of the program were not tested, there was no evolutionary pressure to purge the harmful edits. Delta debugging eliminates these edits.

[9] `http://wiki.openwrt.org/doc/uci/uhttpd`

[10] The number of difference locations are counted as the number of unified diff hunks calculated using the `diff` command with the `-u` option.

6.4 Discussion

The results presented here open up the possibility that end users could repair software vulnerabilities in closed source software without special information or aid from the software vendor. We hope that the tooling published with this work,[11] encourages users to patch important vulnerabilities quickly and researchers to release patches simultaneously with exploit announcements.

There are several caveats associated with this initial work. First, we demonstrated repair on a single executable, and it is possible that the success in the absence of regression test suite will not generalize. However, our results do not appear to be based on any property unique to the NETGEAR exploits. We conjecture that our success at finding functional repairs in this setting is due to the beneficial impact of minimization and to software mutational robustnesss (Section 3.3). Although we did not test our repairs on physical NETGEAR WNDR3700 hardware, we are confident that they would have the same effect on hardware as they do in emulation.

Whenever a patch is distributed there a risk of someone reverse-engineering an exploit from the patch text [22]. As shown in Table 6.1 our technique sometimes generates patches that are not directly relevant to the repaired vulnerability. It may be possible to avoid this risk by generating obfuscated patches in cases where a regression test suite *is* available and minimization is not performed.

[11] `https://github.com/eschulte/netgear-repair`

Chapter 7

Application: Optimizing Nonfunctional Program Properties

The applications presented in prior chapters have considered only functional properties of software. This chapter describes an application of evolutionary exploration of neutral networks to the optimization of nonfunctional software properties, and find the fitness landscapes defined by these nonfunctional properties to be suitable for evolutionary guided search (as predicted in Section 3.6.4). We present a general Genetic Optimization Algorithm (GOA),[1] and demonstrate its effectiveness.

Runtime requirements for software are increasingly dominated by complex non-functional properties. In some server environments memory footprint and the resultant impact on concurrently running processes is of utmost importance [105], while in other settings minimizing off-chip communication is paramount. At the extremes of very small embedded systems and very large data-centers minimizing energy minimization is more important that runtime efficiency [21]. Data-centers are estimated to have consumed over 1% of the global electricity production in 2010 [88], so techniques for minimizing the energy consumption of software could have immediate

[1]`https://github.com/eschulte/goa`

global impact. Despite the pressing need, existing compilers do not target energy consumption. For example a 2010 bug report against the LLVM compiler suite requesting an `-OE` flag to minimize energy consumption[2] was marked as invalid with a note to use runtime minimization.[3]

Runtime properties such as energy consumption are often the result of complex interactions with the particulars of the hardware and environment in which the software is running, limiting the effectiveness of general techniques. Given the wealth of complex runtime properties of software, the large number of available hardware platforms and configurations, and the impact of innocuous environmental factors such as environment variables [115], the resulting cross-product of potential optimizations (each of which may require individual program transformation, implementation or configuration) far exceeds the resources of compiler developers.

This chapter describes GOA, which is a *post-compilation* optimization technique leveraging evolutionary search to automatically find machine-, environment- and workload-specific optimizations in the space of assembler code programs. The remainder of this chapter introduces GOA (Section 7.1). Its effectiveness is evaluated by reducing energy consumption for the PARSEC [17] benchmark applications on two different hardware platforms. GOA finds both hardware- and workload-specific optimizations, and reduce energy consumption of the PARSEC benchmarks by 20% on average as compared to the most efficient available compiler optimizations (Section 7.2).

This work appeared in ASPLOS 2014 [142].

[2]`http://llvm.org/bugs/show_bug.cgi?id=6210`

[3]The LLVM compiler still does not provide any option for energy minimization.

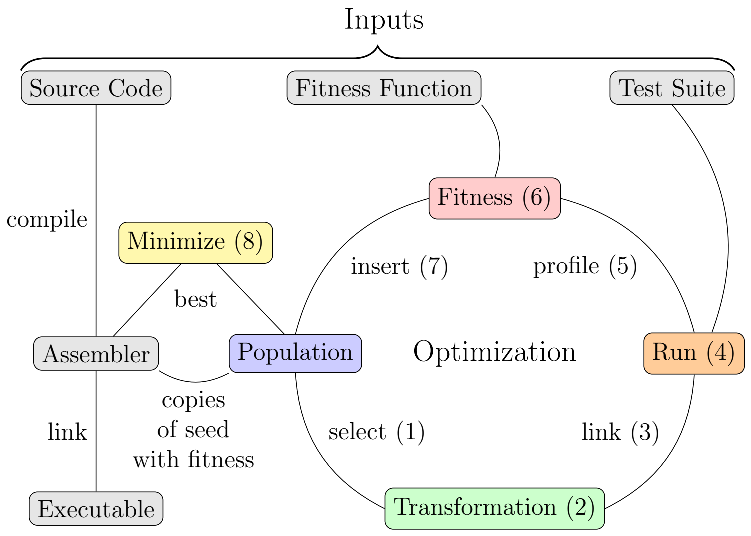


Figure 7.1: Overview of the genetic optimization algorithm.

7.1 Genetic Optimization Algorithm

GOA is a post-compilation, workload-driven optimization technique. GOA takes as input the assembler code produced by an optimizing compiler such as `GCC`. This input is parsed into an ASM-level representation (Section 3.1). GOA uses workloads provided by the software developer to exercise candidate optimizations both to evaluate candidate optimizations both for functionality and to measure runtime properties. An overview of GOA is given in Figure 7.1.

The ASM-level representation of the original program is extracted from the build process, assigned a fitness (illustrated in steps 4, 5, 6 of Figure 7.1) and used to seed a population of program variants. An evolutionary computation algorithm (Section 7.1.2) then searches for candidate program optimizations. Every iteration of the main search loop: (1) selects a candidate optimization from the population, (2) transforms it (Section 3.1.2), (3) links the result into an executable (Section 3.2.1), (4) runs the resulting executable against the supplied workload (as in Section 3.2.2), (5) collects performance information for programs that correctly process the workload, (6) combines the profiling information into a scalar fitness score using the fitness function, and (7) reinserts the optimization and its fitness score into the population. The process continues until either a desired optimization target is reached or a predetermined time budget is exceeded. When the algorithm completes, a post-processing step (8) takes the best individual found in the search and minimizes it with respect to the original program (as in Section 6.3.2.3). GOA returns an assembler diff which may be either manually reviewed or applied to the original program to produce an optimized version of the program.

7.1.1 Inputs

As shown in Figure 7.1, GOA requires three inputs from the developer, the assembly code of the program to be optimized, a regression test suite that captures required functionality, and a measurable optimization target.

The program to be optimized is presented as a single assembly file, which can either be extracted from the build process, e.g., using `gcc`'s "–combine" flag for C or in other cases, manual concatenation of multiple `.s` assembler files may be required. In practice this was straightforward. Only visible assembler code included in the input will be available to be optimized, so performance-critical library functions must be included inline.

The input workload serves as an implicit specification of correct behavior; a candidate optimization that generate the correct result on this input is assumed to retain all required functionality. Of course there is a risk that optimizations inadequately evaluated by the workload break program behavior, which we address in Section 7.2.2. A post-processing minimization step (implemented as in Section 6.3.2.3) removes most harmful mutations which are not caught by the test suite.

```
Input: Original Program, P : $Program$
Input: Workload, Run : $Program \rightarrow ExecutionMetrics$
Input: Fitness Function, Fitness : $ExecutionMetrics \rightarrow \mathbb{R}$
Parameters: $PopSize$, $CrossRate$, $TournamentSize$, $MaxEvals$
Output: Program that optimizes Fitness
 1: let $Pop \leftarrow PopSize$ copies of $\langle \mathsf{P}, \mathsf{Fitness}(\mathsf{Run}(P)) \rangle$
 2: let $EvalCounter \leftarrow 0$
 3: repeat
 4:    let $p \leftarrow$ null
 5:    if Random() $< CrossRate$ then
 6:       let $p_1 \leftarrow \mathsf{Tournament}(Pop, TournamentSize, +)$
 7:       let $p_2 \leftarrow \mathsf{Tournament}(Pop, TournamentSize, +)$
 8:       $p \leftarrow \mathsf{Crossover}(p_1, p_2)$
 9:    else
10:       $p \leftarrow \mathsf{Tournament}(Pop, TournamentSize, +)$
11:    end if
12:    let $p' \leftarrow \mathsf{Mutate}(p)$
13:    $\mathsf{AddTo}(Pop, \langle p', \mathsf{Fitness}(\mathsf{Run}(p')) \rangle)$
14:    $\mathsf{EvictFrom}(Pop, \mathsf{Tournament}(Pop, TournamentSize, -))$
15: until $EvalCounter \geq MaxEvals$
16: return Minimize(Best($Pop$))
```

Figure 7.2: High-level pseudocode for the main loop of GOA.

Finally, the developer must supply a fitness function which GOA will attempt to optimize (usually minimize). In the current implementation this function must produce a single scalar fitness value. In our demonstration example the fitness function estimates energy consumption from profile data collected during the execution of candidate optimizations. However, any function producing a scalar value could be used, e.g., a valid fitness function could eschew profile data and simply measure the size of the executable. We view the generality of possible fitness functions as a strength of this technique.

7.1.2 The GOA Algorithm

GOA uses a *steady state* evolutionary algorithm (shown in Figure 7.2). This differs from previous applications of evolutionary computation techniques to real-world software (e.g., Genprog) which typically use generational genetic algorithms [95, 96]. Instead of replacing the population in discrete steps (generations), steady state evolutionary algorithms operate on single individual candidate optimizations performing the selection, transformation, evaluation and insertion (steps 1, 2, 4 5 6, and 7 in Figure 7.1 respectively) on single individuals in turn. The benefits of steady state genetic algorithms are presented in Section 2.2.2.

High-level pseudocode for the GOA algorithm is given in Figure 7.2. The main loop is parallelized across multiple threads. Synchronization between threads is only required during access to the population *Pop* and the evaluation counter *EvalCounter*.

The population is initialized with a number of copies of the original program (line 1). In every iteration of the main loop (lines 3–15) the search space of possible optimizations is explored by transforming the program using random mutation and crossover operations (described in the next subsection). The probability *CrossRate* controls the application of the crossover operator (lines 6–8). If a crossover is to be performed, two high-fitness parents are chosen from the population via *tournament selection* [130, Section 2.3] and combined to form one new optimization (line 8). Otherwise, a single high-fitness optimization is selected. In either case, the candidate optimization is mutated (line 12), its fitness is calculated (by linking it and running it on the test suite, see Section 3.2), and it is reinserted into the population (line 13). The steady state algorithm then selects a member of the population for eviction using a "negative" tournament to remove a low-fitness candidate and keep the population size constant (line 14). The Fitness function in Figure 7.2 penalizes those variants that fail any test case with an infinitely bad fitness value, and they are quickly purged from the population. Eventually, the fittest candidate optimization is identified, minimized to remove unnecessary or redundant changes (Section 6.3.2.3), and is returned as the result.

7.2 Evaluation

We empirically evaluate both the ability of GOA to reduce energy consumption across multiple population benchmark applications and hardware platforms, and we evaluate the degree to which these optimizations retain program functionality.

We evaluate the effectiveness of GOA against the popular PARSEC benchmark applications (Section 7.2.1). In addition to the program assembler, GOA requires a fitness function and characteristic test suite. We develop an energy model which is appropriate for use as a fitness function (Section 7.2.3). The PARSEC applications include multiple tests, from which we choose the smallest test which produces a runtime of at least one second on each hardware platform. After running GOA, we evaluate the evolved optimizations using physical wall-plug energy measurements.

7.2.1 Benchmarks

Program	C/C++ LOC	ASM LOC	Description
blackscholes	510	7,932	Finance modeling
bodytrack	14,513	955,888	Human video tracking
ferret	15,188	288,981	Image search engine
fluidanimate	11,424	44,681	Fluid dynamics animation
freqmine	2,710	104,722	Frequent itemset mining
swaptions	1,649	61,134	Portfolio pricing
vips	142,019	132,012	Image transformation
x264	37,454	111,718	MPEG-4 video encoder
total	225,467	1,707,068	

Table 7.1: Selected PARSEC benchmark applications.

We use the PARSEC [17] benchmark suite of programs representing "emerging workloads." We evaluate GOA on all of the PARSEC applications that produce testable output and include more than one input set. Testable output is required to ensure that the optimizations retain required functionality. Multiple input sets are required because we use one (training) input set during the GOA optimization and separate held-out ("testing") inputs to test after GOA completes its optimization (Section 7.2.2). The eight applications satisfying these requirements are shown with their sizes and brief descriptions in Table 7.1. Two PARSEC applications were excluded because they did not support this experimental design: `raytrace` which does not produce any testable output, and `facesim` which does not provide multiple input sets.

We evaluate on Intel Core i7 and AMD Opteron machines. The Intel system has 4 physical cores, Hyper-Threading, and 8 GB of memory, and it is indicative of desktop or personal developer hardware. The AMD system has 48 cores and 128 GB of memory, and is representative of more powerful server-class machines.

We compare the performance of GOA's optimized executables to the original executable compiled using the PARSEC tool with its built-in optimization flags or the `gcc` "–Ox" flag that has the least energy consumption.

7.2.2 Held-Out Tests

We use a large held-out test suite to evaluate the degree to which the optimizations found by GOA customize the program semantics to the training test and therefore lose generality. For each benchmark besides `blackscholes`, we randomly generate 100 sets of command-line arguments from the valid flags accepted by the program. The `blackscholes` application accepts no flags, but does read an input file containing a number of financial records. We generated 100 test input files for `blackscholes` by randomly sampling between 2^{14} and 2^{20} records from the set of all records that appear in the multiple PARSEC `blackscholes` tests.

Each test was run using the original program and its output as an oracle to validate the output of the optimized program. We only use combinations of flags which are accepted by the original program, and for which the original program consistently generates the same output.

We evaluate optimized programs in Section 7.2.5 by comparing their outputs against the oracle output. In most cases, we used a binary comparison between output files. However, for `x264` tests producing video output, we find binary exactness overly constraining and instead use manual visual comparison to determine output correctness.

7.2.3 Energy Model

Our fitness function uses a linear energy model based on process-specific hardware counters similar to that developed by Shen *et al.* [150]. We simplify their model in that we:

- We do not build workload-specific power models. Instead, we develop one power model per machine trained to fit multiple workloads and use this single model for every benchmark on that machine.

- We do not consider shared resources, instead we only augment performance-based terms with a single constant base C_{const} energy draw for the machine.

The linear energy model shown in Equation 7.1 combines the hardware counters described in Table 7.2 into a scalar estimate of energy consumption.

$$\frac{energy}{time} = C_{const} + C_{ins}\frac{ins}{cycle} + C_{flops}\frac{flops}{cycle} + C_{tca}\frac{tca}{cycle} + C_{mem}\frac{mem}{cycle} \tag{7.1}$$

The trained values for the model coefficients are given in Table 7.2. They were obtained empirically for each target architecture, using data collected across multiple execution of each PARSEC benchmark, the SPEC CPU benchmark suite, and the `sleep` UNIX utility. For each program, we collected the performance counters as well as the average Watts consumed, as measured by a physical wall plug meter. We combined these data in a linear regression to determine the coefficients shown in Table 7.2.

Coefficient	Description	Intel (4-core)	AMD (48-core)
C_{const}	constant power draw	31.530	394.74
C_{ins}	instructions	20.490	-83.68
C_{flops}	floating point ops.	9.838	60.23
C_{tca}	cache accesses	-4.102	-16.38
C_{mem}	cache misses	2962.678	-4209.09

Table 7.2: Power model coefficients.

The coefficients we obtained differ significantly between the two architectures. The disparity between the AMD and Intel coefficients is likely explained by significant differences in the size and class of the two machines. For example, the $13\times$ increase in idle power of the AMD machine as compared to the Intel machine is reasonable given the presence of 12 times as many cores, and 15 times as much memory.

Even without our simplifications, the predictive power of linear models is rarely perfect. McCullough *et al.* note that on a simple multi-core system, CPU-prediction error is often 10–14% with 150% worst case error prediction [109]. We checked for the presence of overfitting using 10-fold cross-validation and found a 4–6% difference in the average absolute error, which is sufficiently accurate to guide our evolutionary search.

	Program Changes				Energy Reduction				Runtime Reduction	
	Code Edits		Binary Size		Training		Held-out		Held-out	
Program	AMD	Intel	AMD	Intel	AMD	Intel	AMD	Intel	AMD	Intel
blackscholes	120	3	-8.2%	0%	92.1%	85.5%	91.7%	83.3%	91.7%	81.3%
bodytrack	19656	3	-38.7%	0%	0%	0%	0.6%	0%	0.3%	0.2%
ferret	11	1	84.8%	0%	1.6%	0%	5.9%	0%	-7.9%	-0.1%
fluidanimate	27	51	-3.3%	11.4%	10.2%	0%	—	—	—	—
freqmine	14	54	18.7%	34.9%	3.2%	0%	3.3%	-1.6%	3.2%	0.1%
swaptions	141	6	27.0%	18.5%	42.5%	34.4%	41.6%	36.9%	42.0%	36.6%
vips	57	66	-52.8%	0%	21.7%	20.3%	21.3%	—	29.8%	—
x264	34	2	0%	0%	8.3%	0%	9.2%	0%	9.8%	0%
average	2507.5	23.3	3.4%	8.1%	22.5%	17.5%	24.8%	19.8%	24.1%	19.7%

Table 7.3: GOA energy-optimization results on PARSEC applications.

We find that our models have an average of 7% absolute error relative to the wall-socket measurements. The overhead of collecting hardware counter values has no noticeable impact on our test suite run time. Thus, our power model is both sufficiently efficient and accurate to serve as our fitness function.

The Intel Performance Counter Monitor (PCM) counter can also be used to estimate energy consumption. We did not use this counter because the model used by the PCM to estimate energy is not public, and because it estimates energy consumption for an entire socket and does not provide the per-process energy consumption required by our technique.

7.2.4 Energy Reduction Results

The results of our energy consumption minimization are shown in Table 7.3. The "Code Edits" column shows the number of unified diffs (as calculated using the GNU diffutils package[4]) between the original and optimized versions of the assembly program. "Binary Size" indicates the change in size of the compiled executable. The "Energy Reduction" columns report the physically measured energy reduction compared to the original required to run the tests in the fitness function ("Training

[4]`http://www.gnu.org/software/diffutils/`

Workload”) or to run all other PARSEC workloads for that benchmark (“Held-Out Workloads”). For example, if the original program requires 100 units and the optimized version requires 20, that corresponds to an 80% reduction. The “Runtime Reduction” columns report the decrease in runtime compared to the original. In some cases, the measured energy reduction is statistically indistinguishable from zero ($p > 0.05$). Note that for some benchmarks (e.g., `bodytrack`), although there is no measured improvement, the minimization algorithm maintains modeled improvement, resulting in a new binary. We do not report energy reduction on workloads for which the optimized variant did not pass the associated tests (indicated by dashes).

GOA found optimizations that reduced energy consumption in many cases, with the overall reduction on the supplied workloads averaging 20%. Although in some cases—such as in `bodytrack` on AMD or `bodytrack`, `ferret`, `fluidanimate`, `freqmine` and `x264` on Intel—GOA failed to find optimizations that reduced energy consumption, it found optimizations that reduce energy consumption by an order of magnitude for `blackscholes` and by almost half for `swaptions` on both systems. We find that CPU-bound programs are more amenable to improvement than those that perform large amounts of disk IO. This result suggests that GOA is likely better at generating efficient sequences of executing assembly instructions than at improving patterns of memory access. Overall, when considering only those programs with non-zero improvement, average energy reduction was 39%. The increased improvement on held-out workloads compared to training workloads was expected given the increased comparative size and runtime of most held-out workloads reducing the impact of startup time on total energy consumption.

In most benchmark programs energy reduction is very similar to runtime reduction (see Columns “Energy Reduction” and “Runtime Reduction”, Table 7.3). This is not surprising given the important role of time in our energy model. However, in some cases (e.g., `ferret`) energy was reduced despite an increase in runtime, and in other cases (e.g., `vips`) energy consumption decreased significantly more than runtime.

Although some optimizations are easily analyzed through inspection of assembly patches (e.g., the deletion of "`call im_region_black`" from `vips` skipping unnecessary zeroing of a region of data), many optimizations produce unintuitive assembly changes that are most easily analyzed using profiling tools. Such inspection reveals optimizations (Section 7.2.6) that run the gamut from removing explicit semantic inefficiencies in `blackscholes`, to re-organizing assembly instructions in `swaptions` in such a way as to decrease the rate of branch mispredictions, to exploring trade offs between re-calculating values or looking them up in memory in `vips`. The AMD versions of `fluidanimate` and `x264` seem to improve performance by reducing idle cycles spent waiting for off-chip resources.

7.2.5 Program Correctness Results

	Functionality on Held Out Tests	
Program	AMD	Intel
blackscholes	100%	100%
bodytrack	92%	100%
ferret	100%	100%
fluidanimate	6%	31%
freqmine	100%	100%
swaptions	100%	100%
vips	100%	100%
x264	27%	100%
average	78.1%	91.4%

Table 7.4: GOA correctness of optimized programs on held out test suites.

Strengths of GOA include its ability to change program semantics and to customize programs to the training machine, environment and workload. Unfortunately these abilities raise the possibility that program optimizations may change program semantics to over-fit the training workload and break program behavior on held-out workloads. We experimentally investigate the severity of this threat by running our optimized program variants against large suites of held-out tests (Section 7.2.2). The results are given in Table 7.4.

We find that the majority of the discovered optimizations fully preserve program behavior across the entire suite of held-out tests. We believe that the post-processing minimization step is largely responsible for this surprising protection of un-tested behavior. This is because the minimization removes all mutations which do not directly benefit the runtime properties of the program on the training workload, having the result of removing most mutations occurring in unexercised portions of the program.

7.2.6 Case Studies

This section describes three examples illustrating different types of energy optimizations found by GOA.

`blackscholes` implements a partial differential-equation model of a financial market. Because the model runs so quickly, the benchmark artificially adds an outer loop that executes the model multiple times. These redundant calculations are not detected by standard static compiler analyses. The validated `blackscholes` optimization returned by GOA discovered and removed the redundant calculation on both AMD and Intel hardware. However, the optimization strategy differed between the two architectures. In the Intel case, a "subl" instruction was removed, preventing multiple executions of a loop, while in the AMD case a similar effect was obtained by inserting a literal address which (due to the density of valid x86 instructions in random data [12]) is interpreted as valid x86 code to jump out of the loop, skipping redundant calculations.

GOA also finds hardware-specific optimizations in `swaptions`; a benchmark which prices portfolios. On AMD systems, GOA reduces the total `swaptions` energy consumption by 42%. We believe this improvement is mostly due to the reduction in the rate of branch miss-prediction. Although it is not practical for general compilers to reason about branch prediction strategies for every possible processor, GOA naturally finds such environment-specific specialized adaptations.

We found that no single edit (or small subset of edits) accounted for this improvement. Rather, many edits distributed throughout the `swaptions` program collectively reduced mispredictions. Typical edits included insertions and deletions of `.quad`, `.long`, `.byte`, etc., all of which change the absolute position of the executing code. Absolute position affects branch prediction when the value of the instruction pointer is used to index into the appropriate predictor. For example, AMD [70, Section 6.2] advocates inserting `REP` before returns in certain scenarios.

Finally, GOA finds unintuitive optimizations. In the `vips` image processing program, it found an optimization that reduced the total energy used by 20.3% on the Intel system. The optimization actually increased cache misses by 20× but decreased the number of executed instructions by 30%, in effect trading increased off-chip communication for decreased computation. This sort of trade-off in resource consumption is something an experienced developer might attempt, and it is encouraging that our technique is able to find an instance of such a trade-off automatically.

7.3 Discussion

This chapter describes Genetic Optimization Algorithm (GOA), an application that searches within the neutral networks defined by functional program properties in order to optimize nonfunctional program properties. GOA is *powerful*, significantly reducing energy consumption beyond the best available compiler optimizations and capable of customizing software to a target execution environment; *simple*, leveraging widely available tools such as compilers and profilers and requiring no code annotation or technical expertise; and *general*, using generic program transformations (Section 3.1) to target multiple measurable objective functions and applicable to any program that compiles to x86 assembly code.

One notable aspect of the optimizations found by GOA is the *size* of the optimizations in terms of mutational distance (as shown in Section 3.5) from the original program (e.g., the `swaptions` optimization reviewed in Section 7.2.6 requires hundreds of disparate edits). In contrast, the absolute size of program modifications resulting from the evolutionary applications presented in Chapters 5 and 6, and prior work such as Genprog is very small. Minimized repairs in Chapter 6 typically are only differ from the original in two bytes. In a review of the repairs found by Genprog in a large systematic study [95], all repairs could be reduced to at most two mutation operations.

This difference may suggest a connection between continuous fitness landscapes, which provide continual feedback to the evolutionary search technique as described in Section 3.6.4, and the power of evolutionary techniques. If such a connection holds for functional properties of real-world software, it may motivate the use of automated techniques of "smoothing" the fitness landscapes defined by existing test suites such as those described in Section 8.2.2.

Chapter 8

Future Directions and Conclusion

This dissertation describes an empirical investigation of the robustness of real-world software in the face of randomized mutations, and characterizes the resulting large neutral networks of in program space. This robustness is both the result of the evolutionary development of the current software development ecosystem and an indication that existing software is amenable to the use of evolutionary tools for software maintenance and improvement.

Techniques were demonstrated that automatically improve software robustness, correctness, and efficiency. This work however only begins to probe the horizon of potential automated evolutionary software engineering techniques. In the long term, our aim is the fulfillment of the ultimate goals of both the software engineering and the evolutionary computation communities: the full automation of most software development tasks.

Such an ambitious long-term goal is very distant from the present state of the art. The remainder of this chapter highlights some challenges and opportunities for near-term work in this area.

8.1 Challenges

Before evolutionary techniques can become a regular part of the software development work-flow, change is needed both in the applications themselves and in the culture and expectations of the developers. While the tools presented herein demonstrate a wide range of applicability and power of transformation they are lacking in two essential areas.

Developer interface: This includes the related areas of semantics preservation, communicating development goals to evolutionary processes, and integration into the software development life-cycle.

- This dissertation does not address issues of rigorously limiting the potential impact of program transformations on program behavior. Wide spread use of these applications might require additional techniques for providing guarantees ensuring the protection of certain program properties or semantics.
- Communicating goals to evolutionary program improvement techniques. This challenge is being addressed by the software testing community. Advances in software testing allowing developers to more easily and rigorously enforce desired program properties through tests will directly transfer to test-driven evolutionary techniques.
- Presentation automatically generated transformation to developers and incorporating evolved transformations into the life-cycle of software. Put another way, collaboration between human and evolutionary drivers of software evolution is not currently supported. This depends on the program representation used and is discussed in greater length in Section 3.6.3.

Novel functionality: This dissertation focuses on the improvement of existing software functionality, either through patching defects and vulnerabilities or by changing nonfunctional runtime properties. Although neutral networks are hypothesized to enable the evolution of new functionality in biological systems, the results presented herein do not yet demonstrate the evolution of truly significant novel functionality in software systems.

8.2 Opportunities

There are a number of “next steps” for this work in a variety of directions, some of which address the specific challenges presented in the previous section. This section details a number of these opportunities for future work.

8.2.1 Verification

A major hurdle to the incorporation of evolutionary techniques into standard software development work-flows is the lack of formal verification of the effects or the limits of evolutionary program transformations. There are a number of options for work in this area, incorporating both emerging and long standing tools. Any of these options could be applied to the applications presented in this work either as a post-processing verification step (for processes with longer running times) or as part of the fitness function used by the evolutionary algorithm (for processes which require less execution time). The remainder of this section touches upon possible candidate tools and techniques broken out by the type of analysis performed; diff analysis, static analysis, and dynamic analysis.

8.2.1.1 Diff analysis

The evolutionary transformations presented in this work are typically presented as standard diffs at the source code, assembler or binary level for the AST, ASM and

ELF representations respectively. There are existing techniques for analyzing diffs for their impact on program semantics [149, 160, Section IV.B].

A potential weakness of the optimization technique presented in Chapter 7 is the possibility of changing the semantics of the optimized program. This weakness could be addressed by developing a tool that iterates through every hunk in the assembler diff, rejecting those that can not be formally proven to preserve semantics. Depending on the efficiency, such a tool could be applied after every mutation during the execution of GOA, or it could be saved and run once as a post-processing step.

There are a number of emerging techniques and models for proving important properties of assembler code [73, 78, 149] some of which can be directly applicable to proving semantic equivalence between sequences of assembler code [137, Section 4.1].

8.2.1.2 Static analysis

Automated tools for the static analysis of software source code or assembler code are widely in research and by software developers. Programs to assess software quality have been in use for decades, for example Stephen Johnson's `lint` [76], which flagged suspicious source code likely to contain bugs, emerged not long after the creation of the first portable C compiler. Modern static analysis tools include commercial products [44], and open source tools [93], looking for bugs in commonly misused patterns [65, 16] or in the flow of data through a program [6, 11]. In addition, code complexity metrics commonly used in industrial settings can be automatically computed from source code [108].

Such tools could be incorporated into evolutionary techniques, either as components of the fitness function or as a final post-process step. In these roles, such tools would ensure that metrics of code quality or complexity improve over the course of a run, or they would provide guarantees of minimum quality scores at the end of a run respectively.

8.2.1.3 Dynamic analysis

In addition to simply testing program output, more sophisticated methods of dynamically assessing software behavior could provide increased confidence in evolved program variants. To select just one example, invariant detection systems such as Daikon [46] could be used to ensure that invariants found in the original program are maintained in evolved variants.

8.2.2 Continuous Functional Evaluation

As discussed in Sections 3.2.2 and 3.6.4, the lack of smooth gradients in the functional fitness landscapes (defined by boolean `PASS` or `FAIL` valued tests) used in this dissertation may be limiting the effectiveness of evolutionary search techniques. The evolution of truly novel functionality might require new fitness functions describing continuous functional fitness landscapes.

This section describes methods for automatically smoothing fitness landscapes to provide guidance to evolutionary search technique along flat portions of the fitness landscape. The promise of fitness functions with increased granularity is demonstrated in Chapter 6, in which ternary `PASS`, `FAIL`, or `ERROR` vulnerability tests compensate for the lack of a regression test suite.

The following two examples illustrate how we might automatically convert existing test suites from boolean `PASS` or `FAIL` valued functions into continuous functions with gradients capable of guiding evolutionary search. An empirical investigation of their feasibility, practical, and impact on performance would be informative.

1. Many test cases evaluate success by computing a diff between program output and oracle output. Such tests typically return a boolean indicating if the results are identical or different. Tests of this form could easily and automatically be

converted to return the degree of difference, e.g., in terms of edit distance or possibly domain-specific metrics such as numeric difference for numeric output (e.g., using tools such as `numdiff`[1]).

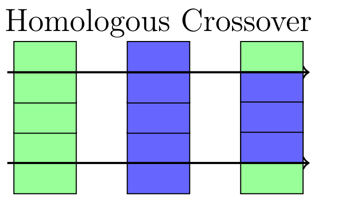
2. In many cases, the output from a test is either successful exit (`ERRNO` equal to 0) or crashing. An alternative would be to compute the number of unique values taken by the program counter during the course of test execution, which might provide an informative gradient to the fitness landscape. Such a metric would indicate how "far" the program gets before crashing, and by limiting the count to unique values of the program counter this test would avoid incorrectly assigning higher fitness infinite loops.

Although these ideas seem to hold promise, they need to be implemented and evaluated. There currently exist large benchmark suites of software defects which are not repairable using the current state of the art automated program repair techniques [95]. Improving the performance of automated repair techniques against such benchmarks would provide a useful metric of the efficacy of these sorts of automated test suite enhancements.

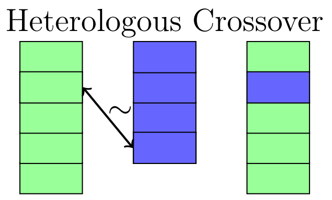
8.2.3 Heterologous Crossover

The re-use of existing software in new projects or contexts is a long standing staple of software development [90, 80]. Moving functionality between different pieces of software is an important intermediate (and arguably sufficient alternative) to the evolution of novel functionality [9]. Crossover is the evolutionary operation responsible for exchanging genetic material between individuals. This section describes extension of the crossover used in this dissertation to allow for genetic material to be shared between heterogeneous software projects.

[1]`http://www.nongnu.org/numdiff`



(a) Homologous Crossover



(b) Heterologous Crossover

Figure 8.1: Homologous and heterologous crossover.

Existing techniques use homologous crossover [52], meaning when crossover is performed a single location is selected and is used between each parent (Figure 8.1 Panel a). This works in homogeneous populations which share a common ancestor and which tend to have equivalent functionality at each location. However, it is not an appropriate technique for sharing functionality between heterogeneous individuals, including; assembler representations compiled using different compiler flags (e.g., `gcc -S` and `gcc -fast`), different major versions of a program, different implementations of a program or even entirely unrelated programs. Therefore an important area of future work is the evolution of a heterogeneous crossover operation capable of transferring functionality between heterogeneous software.

One possible approach would replace the use of crossover points at offsets in the genome, with a search for crossover locations in each parent that share a similar syntactic context (Figure 8.1 Panel b). Some initial work on related techniques in the evolutionary computation community shows promise (e.g., through the use of common sub-sequences between parents as crossover points [68]). Effective heterologous crossover techniques could lead to the sharing optimizations and functionality between different compiler optimizations for a single software project, or even sharing information between different software projects.

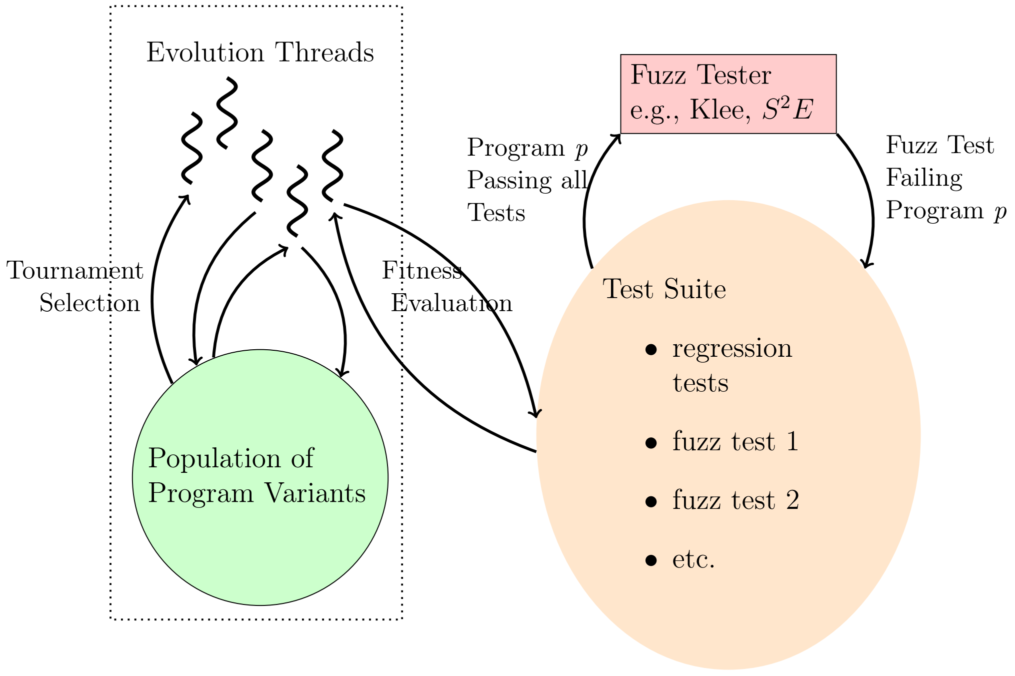


Figure 8.2: Overview of a proposed system for hardening software through the iterative execution of an automated technique for exploit generation and an automated evolutionary technique for defect repair.

8.2.4 Evolutionary Hardening

Binary executables of unknown or proprietary provenance can place significant holes in otherwise secure or trusted validated computer systems. Even on unvalidated platforms such as desktop machines, binary drivers often open significant security vulnerabilities.

Recent advances in symbolic and concolic execution have produced multiple tools for the automated testing of software [26], and even testing of black-box binary executables without access to the source code [29]. Such tools can be used to automatically find tests indicating vulnerabilities in black-box binaries. A system which pairs such an automated test generation tool, with an automated program repair tool such as the ELF-level evolutionary program repair (Chapter 5) could be used for hardening of black-box binary executables.

Figure 8.2 presents a schematic for how such a tool would operate. In this scenario, the technique would begin with a (possibly empty) regression test suite. The binary executable would then iteratively pass between the test generation engine (labeled "Fuzz Tester" in Figure 8.2) yielding a new test indicating a vulnerability in the program, which would then be added to the test suite augmenting the fitness function of the continually running evolutionary program repair process (represented as a dotted box in Figure 8.2). Whenever a version of the program is found that passes all tests, this "repaired" version would be passed back to the test generation engine and the process iterated.

Using such a process to automatically harden black-box executables to the limits of the test generation or program repair engine could greatly improve the security and robustness of many software systems. Initial experiments using simple fuzz testing engines show promise.[2]

8.3 Conclusion

Over the past fifty years software developers have been selecting, reusing and modifying software development tools, code, and design patterns. This history of technological development, through a process mirroring natural selection, may have produced software with the surprisingly biological features that were illuminated in this work, including software mutational robustness, large software neutral networks, and the amenability of extant software to improvement through automated evolutionary processes.

This work establishes that real-world software is amenable to modification through randomized program transformations. This discovery may help explain the recent success of evolutionary software repair techniques (cf. *Genprog* [96]), and it

[2]`https://github.com/eschulte/fuzz-hardening`

contradicts many of the assumptions of the most closely related prior work in this area (cf. *mutation testing* [74]). This discovery opens the door to the application of evolutionary techniques to the automation of many common software development tasks, including the examples given in Chapters 4, 5, and 7, which improve software robustness, correctness and performance. The tools, techniques, and analysis developed to support this work will hopefully establish the foundation for further practical work, unify the fields of evolutionary computation and software engineering, and lead towards the eventual fulfillment of their mutual goal: the increased automation of software development.

Appendices

Appendix A

Software Tools

This chapter describes the software tools used in this work in enough detail to support reproduction and extension of our results [23, 110]. All software developed as part of this dissertation is freely available under open-source licensing. Section A.1 describes the original Genprog software used in Chapters 3, 4, and 5. Section A.2 describes the software evolution library used in Chapters 3, 6, and 7, and describes command line drivers for mutation at the Cil, Clang and LLVM levels. Section A.3 describes the tooling used to patch the closed source NETGEAR binary in Chapter 6. Section A.4 describes the GOA implementation, used in Chapter 7.

A.1 Genprog

An implementation of the Genprog automated evolutionary software repair technique is available online.[1] Genprog version 1.0 was used for the initial mutational robustness experiments presented in Chapter 3. Instructions for usage should are available alongside the source. My early work on the ASM and ELF program representations (Chapter 5 and Sections 3.3.2.1 through 3.3.2.4, and 3.4.1) was implemented within

[1]`http://dijkstra.cs.virginia.edu/genprog`

this software framework. However, later work (Sections 3.3.2.5, and 6 and Chapter 7) uses the software evolution library described in Section A.2.

A.2 Software Evolution Library

The SOFTWARE-EVOLUTION library enables the programmatic modification and evaluation of extant software. The library defines a generic API which abstracts over multiple program representation backends, and provides higher-level functions for evolutionary program modification which make use of this API. The library is implemented in common lisp and should be adaptable to any ANSI common lisp implementation [4]. The library was tested and is known to work with Steel Bank Common Lisp (SBCL) and clozure Common Lisp (CCL). Documentation is provided in the software evolution manual [139], which is available online.[2] The implementation is available online,[3] and can be installed using the QuickLisp[4] Common Lisp package management system.

The structure of the SOFTWARE-EVOLUTION library is shown in Figure A.1. A common interface defines an abstraction over multiple software representations and provides a uniform set of methods for program transformation and evaluation. Methods supporting program modification are implemented on top of these general methods including both evolutionary and Markov Chain Monte Carlo (MCMC) search techniques. MCMC techniques have found recent use in automated techniques of software optimization [137].

The SOFTWARE-EVOLUTION library requires the following dependencies which were also implemented as part of this dissertation. The ELF library for the

[2]`http://eschulte.github.io/software-evolution`
[3]`https://github.com/eschulte/software-evolution`
[4]`http://www.quicklisp.org/beta`

```
                                                     population functions
global variables                                     --------------------
----------------        +-------------------+        incorporate
*population*            |   *population*    |        evict
*max-population-size*   |-------------------|        tournament
*tournament-size*       |      list of      |        mutate
*fitness-predicate*     | software objects  |        new-individual
*cross-chance*          +-------------------+        evolve
*fitness-evals*                   |                  mcmc
*running*                       +-+-+
                                | | |                software functions
                        +-------------------+        --------------
evolve arguments        | software object   |        genome
----------------        |-------------------|        phenome
max-evals               | edits,            |        copy
max-time                | fitness           |        pick-good
target                  | ...               |        pick-bad
period                  +-------------------+        mutate
period-func                        |                 crossover
filter                             |
              +---------------+---+------------+----------------+
              |               |                |                |
     +---------------+  +--------------+  +--------------+  +-------------+
     |      AST      |  |     ELF      |  |     lisp     |  |     asm     |
     |---------------|  |--------------|  |--------------|  |-------------|
     |   Abstract    |  | Executable   |  | lisp source  |  |  assembly   |
     |  Syntax Tree  |  |  Linkable    |  +--------------+  |    code     |
     +---------------+  |   Format     |                    +-------------+
             |          +--------------+                          |
    +--------------+-------------------+                  +-------------------+
    |              |                   |                  |     asm-range     |
 +-------+  +-----------------+    +----------+           |-------------------|
 | Clang |  |       CIL       |    |   LLVM   |           | memory efficient  |
 |-------|  |-----------------|    |----------|           +-------------------+
 | C AST |  | C Intermediate  |    | LLVM IR  |
 +-------+  |    Language     |    +----------+
            +-----------------+
```

Figure A.1: High-level design of the software evolution library.

programmatic manipulation of ELF files.[5] The DELTA-DEBUG library for programmatic and command line delta debugging.[6]

Optional external command line drivers are used for the Cil, CLang, and LLVM program representations, these are listed in Table A.1. These drivers provide uniform interfaces to these diverse external tools.

[5]`https://github.com/eschulte/elf`

[6]`https://github.com/eschulte/delta-debug`

Tool	URL
Cil [116]	`https://github.com/eschulte/cil-mutate`
CLang [92]	`https://github.com/eschulte/clang-mutate`
LLVM [93]	`https://github.com/eschulte/llvm-mutate`

Table A.1: Command line executables implementing program mutation.

A.3 NETGEAR Repair

Full reproduction instructions,[7] code, and tooling used to perform the NETGEAR binary repair are available online.[8] These tools can be used to automatically change the behavior of other binary ELF executables, making it possible to customize and alter binary executables independent of the software's developer.

A.4 Genetic Optimization Algorithm

The implementation of the Genetic Optimization Algorithm (GOA) introduced in Chapter 7 is available online.[9] The implementation can be compiled to a command line executable capable of optimizing user-specified nonfunctional fitness functions. Detailed installation and usage instruction are provided with the program source and in the `README.md` file in the source code repository.

GOA is implemented using the SOFTWARE-EVOLUTION library (Section A.2).

[7]`http://eschulte.github.io/netgear-repair/INSTRUCTIONS.html`

[8]`https://github.com/eschulte/netgear-repair`

[9]`https://github.com/eschulte/goa` — the now outdated version used to generate the experimental results presented in Chapter 7 is preserved at `https://github.com/eschulte/goa/tree/asplos2014`

Appendix B

Data Sets

The experiments described in this work make use of a number of suites of benchmark programs. This Appendix lists the benchmark programs used in this work, with links at which they can be obtained and pointers to where they were used herein.

- A benchmark collection of open source systems programs is available online.[1] These benchmark programs are used in Section 3.3 and in Chapter 4. The raw experimental data presented in Chapter 3.3 is also available.[2]

- The Siemens Software-artifact Infrastructure Repository[3] is used in experiments in Section 3.3.

- Multiple sorting algorithms originally from Rosetta Code[4] are available online[5] along with a complete test suite and source code required to reproduce a number of experiments. These sorting algorithms are used in Chapter 3.

[1] `https://cs.unm.edu/~eschulte/repro/robustness.tar.bz2`
[2] `https://cs.unm.edu/~eschulte/repro/robustness-results.tar.bz2`
[3] `http://sir.unl.edu`
[4] `http://rosettacode.org`
[5] `https://github.com/eschulte/sorters`

- Embedded Repair benchmark programs used in Chapter 5 are available online.[6] The raw experimental data presented in Chapter 5 is also available.[7]
- The NETGEAR firmware used in Chapter 6 can be downloaded directly from the NETGEAR website,[8] however an archived version is also available.[9]
- The PARSEC 3.0 benchmark applications used in Chapter 7 are available online.[10] Additionally, the tooling used to perform the experiments described in Chapter 7, including downloading and unpacking benchmarks, is available online.[11]

[6] `https://cs.unm.edu/~eschulte/repro/embedded.tar.bz2`

[7] `https://cs.unm.edu/~eschulte/repro/embedded-results.tar.bz2`

[8] `http://www.downloads.netgear.com/files/GDC/WNDR3700V4/WNDR3700V4_V1.0.1.42.zip`

[9] `https://github.com/eschulte/netgear-repair/blob/master/stuff/WNDR3700V4_V1.0.1.42.zip`

[10] `http://parsec.cs.princeton.edu/`

[11] `https://github.com/eschulte/goa/tree/asplos2014`

Glossary

ARM Family of reduced instruction set architectures developed by the British company “ARM Holdings” 85

ASE Automated Software Engineering 4, 75

ASM assembler 9, 12, 17, 18, 20–23, 25, 27, 29, 31, 37, 39–42, 46, 51, 53–55, 57–60, 66, 74, 75, 78–83, 103, 119, 128

ASPLOS Architectural Support for Programming Languages and Operating Systems 4, 75, 102

AST Abstract Syntax Tree 10, 15, 17–20, 22, 25, 27, 31–33, 37, 38, 41, 51, 52, 54, 55, 65, 66, 70, 74, 75, 78–83, 119

CCL clozure Common Lisp 129

CIL C Intermediate Language 15, 18, 20, 25, 27, 31, 40, 41, 52, 66, 70, 74, 78–81, 83, 130, 131

CISC Complex Instruction Set Computer 21, 55

CLang C Language family frontend for LLVM 18, 19, 25, 27, 40, 41, 65, 130, 131

crossover Genetic transformation combining the genetic material from two genotypes to produce at least one new genotype. 10, 17, 18, 21, 23, 88–91, 96, 106, 122, 123

delta debugging a technique of systematically narrowing down a set while maintaining a specified property. Initially developed to isolate a minimal failure-inducing input to identify bugs in software. 16, 92, 99, 130

drift the change in genetic material in a population under random sampling 8, 43, 46

ELF Executable and Linkable Format 9, 18, 21, 22, 25, 27, 40, 41, 66, 74, 75, 78–81, 83–85, 88–91, 97, 99, 120, 124, 128, 130

environmental robustness Robustness of phenotype to changes in the environment 6

fault localization the process of determining the location of program faults 74, 75, 77, 83, 89, 97, 98

fitness landscape A space used to visualize multiple genotypes and their associated fitness 7, 11, 18, 62, 67, 101, 116, 121, 122

functional Functionality of software as a function from inputs to behavior and outputs 27–29, 67, 101, 115, 116, 121

GECCO the conference on Genetic and Evolutionary Computation 16

Genprog Automated evolutionary program repair technique 14–16, 85, 89, 92, 97, 106, 116, 125, 128

GOA Genetic Optimization Algorithm 101–108, 112–116, 120, 128, 131

GPEM Genetic Programming and Evolvable Machines 3, 4, 18, 69

ICSE the International Conference on Software Engineering 16

IR Intermediate Representation 12, 13, 17, 18, 20, 22, 23, 25, 51, 66

ISA Instruction Set Architecture 21

LLVM Low Level Virtual Machine 18–20, 22, 23, 25, 27, 40, 41, 51, 55, 66, 102, 130, 131

MCMC Markov Chain Monte Carlo 129

MIPS Family of reduced instruction set architectures, originally an acronym for "Interlocked Pipeline Stages". 21, 53, 88, 91, 95

mutation A random transformation of genetic material 1, 6–8, 10, 18, 21, 23, 32, 34, 42, 43, 45–48, 50, 57, 61, 65, 66, 70, 80, 88, 89, 92, 104, 114, 117

mutation testing Test suite coverage metric based on the ability to detect program variants (mutants). 14, 62, 63, 65, 72, 73, 126

mutational robustness Robustness of phenotype to changes in the genotype 2, 4, 6, 8, 17, 34, 43, 74

natural selection Process by which genetic traits become more or less frequent in a population as a function of their phenotypic effects on reproduction 1, 2, 8, 10, 61, 125

neutral network Connected network in program space. Nodes in this network are neutral variants, nodes are connected by an edge if one may be reachable by applying a single mutation to the other 1, 2, 4, 7, 8, 17, 18, 45, 60, 61, 68–70, 115, 117, 119, 125

neutral variant A variant which retains required phenotypic functionality present in an original ancestor 29, 45, 54, 60, 64, 65, 68–70, 72

nonfunctional Runtime properties of software execution not directly specified by functional properties 28, 29, 67, 101, 115, 119, 131

program space The space of possible programs defined by a program representation and transformations 47, 48, 51, 52, 54, 55

RISC Reduced Instruction Set Computer 21

SBCL Steel Bank Common Lisp 129

SBST Search-Based Software Testing 16

software mutational robustness Mutational robustness of software 1, 18, 29, 30, 32, 34, 40, 41, 43, 54, 60, 100, 125

specification Requirements of program execution and behavior. Specifications may be written or implied and may be either formally stated in mathematical or programmatic terms or informally stated. 1, 4, 22, 28, 29, 37–39, 65, 104

SSA Static Single Assignment 20

steady state an variation of the traditional genetic algorithm in which no explicit generations are used 10, 93, 96, 106

variant An instance of software which has been changed through the application mutation operations to some original program 15, 28–33, 38, 39, 42–44, 46, 66, 68–71, 83, 89, 92, 106

x86 Family of complex instruction set architectures based on the Intel 8086 CPU. 9, 21, 22, 85, 114, 116

References

[1] David Ackley. personal communication, 2000.

[2] Allen Troy Acree. *On Mutation.* PhD thesis, Georgia Institute of Technology, 1980.

[3] Christoph Adami, Charles Ofria, and Travis C. Collier. Evolution of Biological Complexity. In *Proceedings of the National Academy of Sciences*, pages 4463–4468, 2000.

[4] American National Standards Institute and Computer and Business Equipment Manufacturers Association. *Draft Proposed American National Standard Programming Language Common LISP: X3.226-199x: Draft 12.24, X3J13/92-102.* pub-CBEMA, pub-CBEMA:adr, jul 1992. July 1, 1992.

[5] L.W. Ancel, W. Fontana, et al. Plasticity, Evolvability, and Modularity in RNA. *Journal of Experimental Zoology*, 288(3):242–283, 2000.

[6] Paul Anderson and Tim Teitelbaum. Software Inspection Using Codesurfer. In *Workshop on Inspection in Software Engineering (CAV 2001).* Citeseer, 2001.

[7] J. Anvik, L. Hiew, and G.C. Murphy. Coping with an Open Bug Repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.

[8] Ashish Arora, Anand Nandkumar, and Rahul Telang. Does Information Security Attack Frequency Increase with Vulnerability Disclosure? An Empirical Analysis. *Information Systems Frontiers*, 8(5):350–362, 2006.

[9] W Brian Arthur. *The Nature of Technology: What it is and how it Evolves.* Simon and Schuster, 2009.

[10] Thomas Back, David B Fogel, and Zbigniew Michalewicz. *Handbook of Evolutionary Computation.* IOP Publishing Ltd., 1997.

[11] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. CodeSurfer/x86—a Platform for Analyzing x86 Executables. In *Compiler Construction*, pages 250–254. Springer, 2005.

[12] E.G. Barrantes, D.H. Ackley, S. Forrest, and D. Stefanović. Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, 2005.

[13] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored Source Code Transformations to Synthesize Computationally Diverse Program Variants. *arXiv preprint arXiv:1401.7635*, 2014.

[14] Jacob Beal and Gerald Jay Sussman. Engineered Robustness by Controlled Hallucination. In *AAAI 2008 Fall Symposium" Naturally-Inspired Artificial Intelligence*, 2008.

[15] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

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

[17] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[18] Z.D. Blount, C.Z. Borland, and Richard E. Lenski. Historical Contingency and the Evolution of a Key Innovation in an Experimental Population of Escherichia Coli. *Proceedings of the National Academy of Sciences*, 105(23):7899, 2008.

[19] James Bornholt, Todd Mytkowicz, and Kathryn S McKinley. Uncertain: a First-Order Type for Uncertain Data. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 51–66. ACM, 2014.

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

[21] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 83–94, 2000.

[22] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157. IEEE, 2008.

[23] Jonathan B Buckheit and David L Donoho. *Wavelab and Reproducible Research*. Springer, 1995.

[24] T.A. Budd and D. Angluin. Two Notions of Correctness and their Relation to Testing. *Acta Informatica*, 18(1):31–45, 1982.

[25] U.S Department of Labor Bureau of Labor Statistics. Computer Software Engineers and Computer Programmers. On the internet, November 2011. http://www.bls.gov/oco/ocos303.htm.

[26] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.

[27] Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 33–52. ACM, 2013.

[28] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolo Perino, and Mauro Pezze. Automatic Recovery from Runtime Failures. In *International Conference on Software Engineering*. IEEE Press, 2013.

[29] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. *ACM SIGARCH Computer Architecture News*, 39(1):265–278, 2011.

[30] S. Ciliberti, O.C. Martin, and Andreas Wagner. Innovation and Robustness in Complex Regulatory Gene Networks. *Proceedings of the National Academy of Sciences*, 104(34):13591, 2007.

[31] S. Ciliberti, O.C. Martin, and Andreas Wagner. Robustness can Evolve Gradually in Complex Regulatory Gene Networks with Varying Topology. *PLoS Computational Biology*, 3(2):e15, 2007.

[32] TIS Committee et al. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. *TIS Committee*, 1995.

[33] C. Cowan, H. Hinton, C. Pu, and J. Walpole. The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *Proc. of the 23^{rd} National Information Systems Security Conference (NISSC)*, Oct 2000.

[34] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stack-Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[35] Zachary Cutlip. Complete, Persistent Compromise of Netgear Wireless Routers, October 2013. http://shadow-file.blogspot.com/2013/10/complete-persistent-compromise-of.html.

[36] Charles Darwin. *On the Origin of Species*, volume 484. John Murray, London, 1859.

[37] Kenneth A De Jong and William M Spears. A Formal Analysis of the Role of Multi-Point Crossover in Genetic Algorithms. *Annals of Mathematics and Artificial Intelligence*, 5(1):1–26, 1992.

[38] J. Arjan G. M. de Visser, Joachim Hermisson, Günter P. Wagner, Lauren Ancel Meyers, Homayoun Bagheri-Chaichian, Jeffrey L. Blanchard, Lin Chao, James M. Cheverud, Santiago F. Elena, Walter Fontana, Greg Gibson, Thomas F. Hansen, David Krakauer, Richard C. Lewontin, Charles Ofria, Sean H. Rice, George von Dassow, Andreas Wagner, and Michael C. Whitlock. Perspective: Evolution and Detection of Genetic Robustness. *Evolution*, 57(9):1959–1972, 2003.

[39] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, 1978.

[40] R.A. Demillo and A.J. Offutt. Constraint-Based Automatic Test Data Generation. *Software Engineering, IEEE Transactions on*, 17(9):900–910, 1991.

[41] Carsten Dominik. *The Org Mode 7 Reference Manual-Organize your life with GNU Emacs*. Network Theory Ltd., 2010.

[42] J.A. Draghi, T.L. Parsons, G.P. Wagner, and J.B. Plotkin. Mutational Robustness can Facilitate Adaptation. *Nature*, 463(7279):353–355, 2010.

[43] Gerald M. Edelman and J.A. Gally. Degeneracy and Complexity in Biological Systems. *Proceedings of the National Academy of Sciences*, 98(24):13763, 2001.

[44] Pär Emanuelsson and Ulf Nilsson. A Comparative Study of Industrial Static Analysis Tools. *Electronic notes in theoretical computer science*, 217:5–21, 2008.

[45] W. EricWong and Vidroha Debroy. A Survey of Software Fault Localization. Technical report, The University of Texas at Dallas, Dallas, TX, USA, 2009.

[46] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[47] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 449–460. IEEE Computer Society, 2012.

[48] M.A. Félix and Andreas Wagner. Robustness and Evolution: Concepts, Insights and Challenges from a Developmental Model System. *Heredity*, 100(2):132–140, 2006.

[49] Dennis Fisher. D-Link Planning to Patch Router Backdoor Bug, October 2013. http://threatpost.com/d-link-planning-to-patch-router-backdoor-bug/102581.

[50] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.

[51] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building Diverse Computer Systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[52] Frank D Francone, Markus Conrads, Wolfgang Banzhaf, and Peter Nordin. Homologous Crossover in Genetic Programming. In *GECCO*, pages 1021–1026, 1999.

[53] Phyllis G Frankl, Stewart N Weiss, and Cang Hu. All-Uses vs Mutation Testing: an Experimental Comparison of Effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[54] Stefan Frei, Bernhard Tellenbach, and Bernhard Plattner. 0-day Patch Exposing Vendors (in) Security Performance. *BlackHat Europe, Amsterdam, NL*, 2008.

[55] Zachary P. Fry and Wes Weimer. A Human Study of Fault Localization Accuracy. In *International Conference on Software Maintenance*, pages 1–10, 2010.

[56] Richard P Gabriel. LISP: Good News, Bad News, how to Win Big. *AI EXPERT.*, 6(6):30–39, 1991.

[57] T.L. Graves, M.J. Harrold, J.M. Kim, A. Porter, and G. Rothermel. An Empirical Study of Regression Test Selection Techniques. *Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[58] B.J.M. Grun, D. Schuler, and A. Zeller. The Impact of Equivalent Mutants. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 192–199. IEEE, 2009.

[59] Mark Harman. personal communication, 2013.

[60] E.J. Hayden, C. Weikert, and A. Wagner. Directional Selection Causes Decanalization in a Group I Ribozyme. *PLOS ONE*, 7(9):e45351, 2012.

[61] John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett, and John Gill. MIPS: A Microprocessor Architecture. *ACM SIGMICRO Newsletter*, 13(4):17–22, 1982.

[62] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and Martin Rinard. Power-Aware Computing with Dynamic Knobs. Technical report, Technical Report TR-2010-027, CSAIL, MIT, 2010.

[63] J.H. Holland. Outline for a Logical Theory of Adaptive Systems. *Journal of the ACM (JACM)*, 9(3):297–314, 1962.

[64] John Henry Holland. *Adaptation in Natural and Artificial Systems: an Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* The MIT press, 1992.

[65] David Hovemeyer and William Pugh. Finding Bugs is Easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[66] William E. Howden. Weak Mutation Testing and Completeness of Test Sets. *Software Engineering, IEEE Transactions on*, pages 371–379, 1982.

[67] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the Effectiveness of Dataflow-and Controlflow-Based Test Adequacy Criteria. In *International Conference on Software Engineering*, pages 191–200, 1994.

[68] Ben Hutt and Kevin Warwick. Synapsing Variable-Length Crossover: Meaningful Crossover for Variable-Length Genomes. *Evolutionary Computation, IEEE Transactions on*, 11(1):118–131, 2007.

[69] Dimitris Iliopoulos, Christoph Adami, and Peter Szor. Darwin Inside the Machines: Malware Evolution and the Consequences for Computer Security. *Virus Bulletin*, pages 187–194, 2008.

[70] Advanced Micro Devices Incorporated. Software Optimization Guide for AMD64 Processors. Technical report, Advanced Micro Devices Incorporated, September 2005. `http://support.amd.com/TechDocs/25112.PDF`.

[71] Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. Compiler-Generated Software Diversity. In *Moving Target Defense*, pages 77–98. Springer, 2011.

[72] Sushil Jajodia, Anup K Ghosh, Vipin Swarup, Cliff Wang, and X Sean Wang. *Moving Target Defense*. Springer, 2011.

[73] Jonas B Jensen, Nick Benton, and Andrew Kennedy. High-Level Separation Logic for Low-Level Code. In *ACM SIGPLAN Notices*, volume 48, pages 301–314. ACM, 2013.

[74] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[75] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated Atomicity-Violation Fixing. *ACM SIGPLAN Notices*, 46(6):389–400, 2011.

[76] Stephen C Johnson. *Lint, a C Program Checker*. Citeseer, 1977.

[77] James A. Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Automated Software Engineering*, pages 273–282, 2005.

[78] Andrew Kennedy, Nick Benton, Jonas B Jensen, and Pierre-Evariste Dagand. Coq: the World's Best Macro Assembler? In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, pages 13–24. ACM, 2013.

[79] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[80] M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 83–92. IEEE, 2004.

[81] M. Kimura. *The Neutral Theory of Molecular Evolution*. Cambridge University Press, 1985.

[82] M. Kimura et al. Evolutionary Rate at the Molecular Level. *Nature*, 217(5129):624, 1968.

[83] K.N. King and A.J. Offutt. A Fortran Language System for Mutation-Based Software Testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[84] H. Kitano. Biological Robustness. *Nature Reviews Genetics*, 5(11):826–837, 2004.

[85] John C. Knight and Paul E. Ammann. An Experimental Evaluation of Simple Methods for Seeding Program Errors. In *International Conference on Software Engineering*, 1985.

[86] R.D. Knight, S.J. Freeland, and L.F. Landweber. Selection, History and Chemistry: the Three Faces of the Genetic Code. *Trends in biochemical sciences*, 24(6):241–247, 1999.

[87] Robert Könighofer and Roderick Bloem. Repair with On-The-Fly Program Analysis. In *Haifa Verification Conference. Springer*, 2012.

[88] Jonathan Koomey. Growth in Data Center Electricity Use 2005 to 2010. *Oakland, CA: Analytics Press. August*, 1:2010, 2011.

[89] John R. Koza. Genetic Programming: On the Programming of Computers by Means of Natural Selection, 1992. *See http://miriad. Iip6. fr/microbes Modeling Adaptive Multi-Agent Systems Inspired by Developmental Biology*, 229, 1992.

[90] Charles W Krueger. Software Reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[91] F. Kühling, K. Wolff, and P. Nordin. A Brute-Force Approac to Automatic Induction of Machine Code on CISC Architectures. *Genetic Programming*, pages 288–297, 2002.

[92] Chris Lattner. LLVM and CLang: Next Generation Compiler Technology. In *The BSD Conference*, pages 1–2, 2008.

[93] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[94] Claire Le Goues. *Automatic Program Repair Using Genetic Programming*. PhD thesis, University of Virginia, 2013.

[95] 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.

[96] 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.

[97] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and Operators for Improving Evolutionary Software Repair. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 959–966. ACM, 2012.

[98] Richard E. Lenski, Jeffrey Barrick, and Charles Ofria. Balancing Robustness and Evolvability. *PLoS biology*, 4(12):e428, 2006.

[99] Richard E. Lenski and M. Travisano. Dynamics of Adaptation and Diversification: a 10,000-Generation Experiment with Bacterial Populations. *Proceedings of the National Academy of Sciences*, 91(15):6808, 1994.

[100] John Levon. *OProfile Manual.* Victoria University of Manchester, 2004.

[101] R Lipton. Fault Diagnosis of Computer Programs. *Student Report, Carnegie Mellon University*, 1971.

[102] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic Input Rectification. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 80–90. IEEE, 2012.

[103] P Lougher and R Lougher. SQUASHFS-A Squashed Read-Only Filesystem for Linux, 2006.

[104] Sean Luke. *Essentials of Metaheuristics.* Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[105] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.

[106] Matias Martinez and Martin Monperrus. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Software Engineering*, pages 1–30, 2013.

[107] Joanna Masel and Meredith V Trotter. Robustness and Evolvability. *Trends in Genetics*, 26(9):406–414, 2010.

[108] Thomas J. McCabe. A Complexity Measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[109] John C McCullough, Yuvraj Agarwal, Jaideep Chandrashekar, Sathyanarayan Kuppuswamy, Alex C Snoeren, and Rajesh K Gupta. Evaluating the Effectiveness of Model-Based Power Characterization. In *USENIX Annual Technical Conf*, 2011.

[110] Jill P Mesirov. Accessible Reproducible Research. *Science*, 327(5964):415–416, 2010.

[111] Lauren Ancel Meyers, Fredric D Ancel, and Michael Lachmann. Evolution of Genetic Potential. *PLoS Comput Biol*, 1(3):e32, 08 2005.

[112] S. Misailovic, D.M. Roy, and Martin Rinard. Probabilistically Accurate Program Transformations. *Static Analysis*, pages 316–333, 2011.

[113] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT press, 1998.

[114] Martin Monperrus. A Critical Review of "Automatic Patch Generation Learned From Human-Written Patches": An Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *Proc. of the Int. Conf on Software Engineering (ICSE), Hyderabab, India*, 2014.

[115] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *ACM Sigplan Notices*, volume 44, pages 265–276. ACM, 2009.

[116] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction*, pages 209–265. Springer, 2002.

[117] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[118] T Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants. *ACM Transactions on Software Engineering and Methodology, to appear*, 2014.

[119] Peter Nordin, Wolfgang Banzhaf, and Frank D Francone. 12 Efficient Evolution of Machine Code for CISC Architectures Using Instruction Blocks and Homologous Crossover. *Advances in genetic programming*, 3:275, 1999.

[120] A.J. Offutt and W.M. Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[121] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[122] A.J. Offutt, Y.S. Ma, and Y.R. Kwon. The Class-Level Mutants of MuJava. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 78–84. ACM, 2006.

[123] A.J. Offutt and J. Pan. Automatically Detecting Equivalent Mutants and Infeasible Paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[124] Charles Ofria, Christoph Adami, and Travis C. Collier. Design of Evolvable Computer Languages. *Evolutionary Computation, IEEE Transactions on*, 6(4):420–424, 2002.

[125] Charles Ofria, W. Huang, and E. Torng. On the Gradual Evolution of Complexity and the Sudden Emergence of Complex Features. *Artificial life*, 14(3):255–263, 2008.

[126] Charles Ofria and Claus O. Wilke. Avida: A Software Platform for Research in Computational Evolutionary Biology. *Artificial Life*, 10(2):191–229, 2004.

[127] Michael Orlov and Moshe Sipper. Genetic Programming in the Wild: Evolving Unrestricted Bytecode. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1043–1050. ACM, 2009.

[128] H Allen Orr. The Genetic Theory of Adaptation: a Brief History. *Nature Reviews Genetics*, 6(2):119–127, 2005.

[129] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 87–102. ACM, 2009.

[130] R. Poli, W.B. Langdon, and N.F. McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises Uk Ltd, 2008.

[131] Etienne Rajon and Joanna Masel. Compensatory Evolution and the Origins of Innovations. *Genetics*, 193(4):1209–1220, 2013.

[132] Martin Rinard. Probabilistic Accuracy Bounds for Fault-Tolerant Computations that Discard Tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334. ACM, 2006.

[133] Martin Rinard. Survival Strategies for Synthesized Hardware Systems. In *Formal Methods and Models for Co-Design, 2009. MEMOCODE'09. 7th IEEE/ACM International Conference on*, pages 116–120. IEEE, 2009.

[134] Martin Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee Jr. Enhancing Server Availability and Security through Failure-Oblivious Computing. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6*, pages 21–21. USENIX Association, 2004.

[135] Martin C Rinard. Living in the Comfort Zone. In *ACM SIGPLAN Notices*, volume 42, pages 611–622. ACM, 2007.

[136] G. Rothermel and M.J. Harrold. Empirical Studies of a Safe Regression Test Selection Technique. *Transactions on Software Engineering*, 24(6):401–419, 1998.

[137] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2013.

[138] David Schuler and Andreas Zeller. (Un-) Covering Equivalent Mutants. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 45–54. IEEE, 2010.

[139] Eric Schulte. *Software Evolution Library*. University of New Mexico, 2014. `http://eschulte.github.io/software-evolution`.

[140] Eric Schulte, Dan Davison, Thomas Dye, and Carsten Dominik. A Multi-Language Computing Environment for Literate Programming and Reproducible Research. *Journal of Statistical Software*, 46(3):1–24, 1 2012.

[141] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13. ACM, 2013.

[142] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler Software Optimization for Reducing Energy. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 639–652. ACM, 2014.

[143] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated Program Repair through the Evolution of Assembly Code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.

[144] Eric Schulte, Zachary. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software Mutational Robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

[145] P. Schuster, W. Fontana, P.F. Stadler, and I.L. Hofacker. From Sequences to Shapes and Back: A Case Study in RNA Secondary Structures. *Proceedings: Biological Sciences*, pages 279–284, 1994.

[146] S. Segura, R.M. Hierons, D. Benavides, and A. Ruiz-Cortés. Mutation Testing on an Object-Oriented Framework: An Experience Report. *Information and Software Technology*, 2011.

[147] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[148] Linda G. Shapiro, George C. Stockman, Linda G. Shapiro, and George Stockman. *Computer Vision.* Prentice Hall, January 2001.

[149] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-Driven Equivalence Checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 391–406. ACM, 2013.

[150] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power Containers: An OS Facility for Fine-Grained Power and Energy Management on Multicore Servers. In *Architectural support for programming languages and operating systems*, pages 65–76, 2013.

[151] TIS Committee, http://x86.ddj.com/ftp/manuals/tools/elf.pdf. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, May 1995.

[152] Michel JG Van Eeten and Johannes M Bauer. Economics of Malware: Security Decisions, Incentives and Externalities. Technical report, OECD Publishing, 2008.

[153] E. Van Nimwegen, James Crutchfield, and M.A. Huynen. Neutral Evolution of Mutational Robustness. *Proceedings of the National Academy of Sciences*, 96(17):9716, 1999.

[154] F.I. Vokolos and P.G. Frankl. Empirical Evaluation of the Textual Differencing Regression Testing Technique. In *International Conference on Software Maintenance*, pages 44–53, 1998.

[155] Andreas Wagner. Neutralism and Selectionism: a Network-Based Reconciliation. *Nature Reviews Genetics*, 9(12):965–974, 2008.

[156] Andreas Wagner. Robustness and Evolvability: a Paradox Resolved. *Proceedings of the Royal Society B: Biological Sciences*, 275(1630):91, 2008.

[157] Andreas Wagner. *Robustness and Evolvability in Living Systems*. Princeton University Press, 2013.

[158] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated Fixing of Programs with Contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.

[159] Westley Weimer. Patches as Better Bug Reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[160] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.

[161] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[162] E.J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 1982.

[163] J. Whitacre and A. Bender. Degeneracy: a Design Principle for Achieving Robustness and Evolvability. *Journal of theoretical biology*, 263(1):143–153, 2010.

[164] Claus O. Wilke, Jia Lan Wang, Charles Ofria, Richard E. Lenski, and Christoph Adami. Evolution of Digital Organisms at High Mutation Rates Leads to Survival of the Flattest. *Nature*, 412(19):331–333, July 2001.

[165] Sewall Wright. *The Roles of Mutation, Inbreeding, Crossbreeding, and Selection in Evolution*, volume 1. na, 1932.

[166] Andreas Zeller. Yesterday, My Program Worked. Today, It Does Not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.

[167] Earl Zmijewski. Reckless Driving on the Internet, February 2009. http://www.renesys.com/2009/02/the-flap-heard-around-the-world/.