Learning from Super-Mutants

Searching post-apocalyptic software ecosystems for novel semantics-preserving transforms

Jason Landsborough
SPAWAR Systems Center Pacific
San Diego, California

Stephen Harding
SPAWAR Systems Center Pacific
San Diego, California

Sunny Fugate
SPAWAR Systems Center Pacific
San Diego, California

ABSTRACT

In light of recent advances in genetic-algorithm-driven automated program modification, our team has been actively exploring the art, engineering, and discovery of novel semantics-preserving transforms. While modern compilers represent some of the best ideas we have for automated program modification, current approaches represent only a small subset of the types of transforms which can be achieved. In the wilderness of post-apocalyptic software ecosystems of genetically-modified and mutant programs, there exist a broad array of potentially useful software mutations, including semantics-preserving transforms that may play an important role in future software design, development, and most importantly, evolution.

CCS CONCEPTS

•Computing methodologies → Genetic algorithms; •Software and its engineering → Empirical software validation; *Software evolution; Search-based software engineering;* **•Security and privacy →** *Software and application security;*

KEYWORDS

Genetic algorithm, genetic improvement, search based software engineering, validation

ACM Reference format:
Jason Landsborough, Stephen Harding, and Sunny Fugate. 2017. Learning from Super-Mutants. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082525

1 INTRODUCTION

Today, software represents a confluence of human ingenuity, diverse and sophisticated libraries, robust compilers and interpreters, and the labor of millions of human programmers. Software evolves primarily as a result of human requirements and manipulation. New software is created, old software is integrated or discarded, and the software *ecosystem* slowly evolves.

This ecosystem is not a natural one. The colloquial term "software ecosystem" seems like a misnomer, and a poor analogy for the blood, sweat, and tears of software development. Indeed, while there exist market pressure and definitive trends in software development, software applications don't evolve over time due to any form of natural selection. With few exceptions, software, once compiled, is static, fixed, and evolution over time one of design and artificial selection pressures. It isn't driven by natural selection or some kind of artificial biology. Different copies of software do not have different "genes," do not undergo natural selection, and do not evolve on their own. At least not yet.

Nonetheless, we find the metaphor a useful one, particularly when describing the nature, purpose, and evolution of malicious programs. Malware authors have used evolutionary principles such as polymorphism and self-modification for several decades. The principles of evolutionary biology, adaptation, co-evolution, and even natural selection, while not yet directly applicable to software environments, still provides a natural language for describing our long-term observations of malware propagation and malware ecologies [5]. Outside of isolated experiments in synthetic biology, a true "software ecosystem" is still just a longstanding dream of the programming adept.

When compared to optimizing compiler-driven software transformation, we have taken a very different approach to automated program modification: mutating programs arbitrarily, performing cursory validation of program behavior, and then extracting and exploring each modification as a candidate transform. This paper describes our research goals with respect to improving the security of programs using "in-place" modification, provides an overview of our technical approach for safe and efficient modification of compiled binary program images, describes our initial results, and discusses the impact that such approaches will have on the future of software ecosystems.

1.1 Why Modification of Binary Files

Software ecosystems notwithstanding, numerous research efforts over the last two decades have been actively exploring software optimization and Genetic Improvement [12, 16, 20, 28], utilizing search-based techniques for purposes as broad-scoped as program optimization [4, 16], energy efficiency [1, 22], program feature modification [15], automated bug fixing [9, 18, 24, 27], feature removal [14], or program improvement [12, 28].

Our research group has recently begun focusing on program binary modification due to the strong desire to be able to modify or otherwise improve programs without access to source code. We have been principally interested in the removal of unwanted, unused, or vulnerable code. We are simultaneously exploring methods for performing instruction-level randomization of programs to protect against code re-use attacks [2, 25]. To these ends we have been seeking methods to perform *in-place* program modification without access to or use of source code, compilers, or the permission or restriction of developers or commercial software vendors.

GECCO '17 Companion, Berlin, Germany
2017. 978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082525

Our current research goal is to create a large set of set of program transformation routines that have a computing cost comparable to traditional program patching. We also desire a larger catalog of potential transforms than what is traditionally available using common techniques (see Section 2.2). While a traditional patch routine overwrites specific program blocks based on differences between two binaries, our desired program transform routines would iteratively (and potentially randomly) apply multiple independent patches to a program to safely affect its size, features, performance, and security-related properties.

Prior work on in-place program binary modification using genetic algorithms has had limited utility for several reasons. While others have demonstrated the feasibility of performing GA-driven program modification, these are generally limited to changes which can be carefully restricted to ensure program safety properties [10, 14, 18]. Ad hoc mutation of software can generally only be performed when safety is not important or when other criteria such as performance are paramount. Indeed, test suites generally provide no guarantee of program reliability, correctness, or safety [21]. For example, removing all unused features from a program in order to limit attack surfaces will also result in removal of any code not specifically affecting program test results. Limiting the changes to a set of specified features, however, can be performed relatively safely [14]. Use of genetic algorithms to modify programs, while convenient in their generality, are also incredibly costly, requiring many hours on a high-performance computing cluster. This approach would be impractical if the goal were to create randomized variants of a particular program. Resulting binaries are likely to be both fault-prone as well as having an inordinately high cost to produce.

We have performed a cursory survey of candidate transforms (learned through observations of a GA population of mutant programs) and compare and contrast these "naturally occurring" transforms to those found in the literature. While many of these transforms do not have the optimizing properties often sought by compiler designers, many of these transforms are suitable to perform automated binary diversification and code removal.

In addition to seeking novel tools to achieve program diversity, we are also seeking to learn novel transforms to achieve other purposes, learning from mutations occurring within synthetic software ecosystem. Our approach and principle contribution to the literature is to demonstrate the feasibility and benefits of an empirical approach to transform discovery. By treating software as an ecosystem and expanding this idea using synthetic evolution, we can learn useful ways of transforming programs to gain novel properties or security benefits.

2 BACKGROUND

Generally, program modification is an unsafe process, resulting in a number of changes which may have unintended side-effects. Modifications which still pass test-cases may still remove, disable, or cripple critical features and safety elements which are not covered by the current test suite. Prior work on GA-based program modification has overcome this difficulty through the use of techniques such as delta-debugging: obtaining a minimal change set for a particular fitness criterion [22, 30]. This approach is relevant when the fitness criterion is explicitly known. For this work, our fitness function's primary purpose was to maintain program validity while allowing the individual populations' members to drift. Essentially, we created a set of neutral variants because we are interested in examining the differences between the variants and the original. Each difference between a mutated population member and the original program becomes a potential *semantics-preserving transformation.*

Modern software is the result of an incredible wealth and history of software-based symbol-manipulation systems [19], the incredible stability and determinism of modern compilers [3], and our trust that these systems are *good enough* [13]. We use automated symbol manipulation systems within our optimizing compilers to achieve enormous performance gains. However, the automated manipulation of software as a symbol system has thus far relied primarily on hand-crafted semantics-preserving transforms. These types of transforms have also been primarily concerned with performance, often at a cost to other potentially useful properties. Unfortunately, it seems that this trust may be misplaced. No matter how carefully we scrutinize the validity of our software, unexpected defects still occur.

As software grows in complexity, developers and researchers have a strong desire to simplify. Traditional approaches to software simplification generally serve to hide complexity using abstraction. We believe there is great benefit to removing complexity from software and in doing so using automated methods.

We have relied on the following assumptions in our research:

(1) Software is generally robust to mutational change [11, 17, 23, 29].
(2) Whole-program validity is a sufficient measure of safety for many situations.
(3) Software is only ever as robust or valid as its test-suite [13, 23].

While each of these appear somewhat specious, we feel that there is value in treating formal software verification with skepticism and pursuing an alternate model for software validation. Treating software verification in this way has allowed us to consider creative methods for mutating and evolving software. Our hope is that this approach will allow us to to extract useful artifacts from the ecosystem of program variants evolved using traditional genetic algorithm approaches.

An astute reader may also ask why do this using a GA rather than just cleverly devising transforms that are known to be useful? Our response is that this, essentially, has already been done. Modern compilers already represent some of the best and most sophisticated techniques available for purposes of program optimization. There are, however, other useful properties that might be gained by *learning from nature* or *learning from the wild.* For some use-cases it might be desirable to use only transforms that make a program smaller, perhaps at the expense of performance. Other use-cases might seek to make the total cost of two parallel paths identical in computational cost even if they perform fundamentally different operations. In essence, we think that there are ample opportunities for the application of an extensive catalog of program transforms. We are also fairly confident that there exist a large number of useful transforms which could play important roles in future software development, construction, deployment, and protection.

2.1 Program Binary Modification

In the literature there are numerous examples of research into in-place modification of binaries and this research encompasses many different methods and goals for the modification:

- To remove unneeded or unwanted features [14].
- To make programs "look" different [6–8].
- To change program performance [1, 4, 15, 22].
- To affect program security properties [6, 7, 14].
- To repair known software defects [24, 27].

Our work has leveraged a broad range of prior research into Genetic Algorithms and in-place binary modification.

Zeller et al. demonstrate an algorithm for automatically reducing program crashes to their minimal input state [30]. Their algorithm, which they dub ddmin (delta debug min), takes an input that produces a crash or failure and repeatedly runs slightly modified versions of the provided input until it has produced the smallest set of input that exhibits the crash. The algorithm uses a memoized binary search to produce output and was demonstrated on multiple programs and input languages. The algorithm was also able to produce a maximal set of failure circumstances when compiling certain defective programs.

Langdon et al. demonstrate that genetic programming techniques can be scaled to work on large, complex pieces of real world software [16]. They introduce a few novel processes, a Sensitivity analysis pass that determines which sections of the program might exhibit the largest improvement in the specified property and a different way of binning output to ease sampling. They also demonstrate that utilizing an automatic test oracle can allow genetic programming to improve on a programs accuracy while also improving on a specific functional property instead of trying to remain true to the original programs semantics.

There's a large body of work that focuses on using genetic algorithms to improve existing software in some form. In their work on using Genetic improvement to reduce energy consumption Bruce et al. explored modifying a program at the source code level to achieve the reduction [1]. Using a fitness function that selected variants based on energy information provided by the processor they found they were able to reduce energy consumption by up to 25% depending on the use case. They also found that for their test program, which was single threaded and CPU bound, that reducing energy use also strongly corresponded with a reduction in execution time.

Conversely in their work using Evolutionary Computing to reduce energy usage Schulte et al. demonstrate a technique that works on programs post compilation and can reduce power consumption by an average of 20% [22]. This reduction was achieved across two different microarchitectures. The reduction is measured using a linear energy model that has been trained across multiple workloads for each architecture and is validated with wall socket measurements. Variants were selected first for their ability to pass a functionality test suite and then selected again based on their energy use. They found that their evolutionary computation technique made a wide range of changes to achieve this reduction in power use. For one program it made numerous small changes where no single change or group of changes could be traced to the reduction. In another case they found it traded an increase in cache miss rate for a decrease in total computation time.

Forrest and Weimar et al. have shown that it is possible to fix bugs in existing C programs through manipulation of of an abstract syntax tree at the statement level [10]. They then extend this work and show that similar techniques can be used to repair a program through the mutation of assembly code as opposed to source [27]. This brings several advantages over their previous work, primarily that it is more generally applicable as it can be used on any language that compiles to an intermediate assembly language instead of requiring source access. It allows a finer granularity of repairs to be effected than what is available from the statement level. Finally they demonstrate the ability to fix vulnerabilities in router firmware before the vendor publicly released a patch [24].

Landsborough et al. show that genetic programming techniques can be used to remove unwanted or unused program functionality while still allowing the variant to pass its test suite successfully [14]. By providing test cases that only contain program features that they wanted removed they were able to successfully remove the desired functionality from a small test program as well as two different utility functions. Recognizing that the genetic algorithm may have removed more program functionality than what they had intended they were able to use delta debugging [30] to create a minimal set of changes that still removed the unwanted functionality.

Research has been performed that examines how brittle source languages and mutations are. Velez et al. look at the robustness of source code in a large corpus of Java projects [26]. By examining the minimal distinguishing subset of functions in their corpus they found that only a very small amount of source code, approximately 4%, is distinctive. They found that the minimal subsets provide a concise window into a programs function and purpose along with the possibility of automatically suggesting code snippets to improve a programmers productivity. They also show that these minimal sets can be used to greatly improve the capability of a programmer to perform effective code search.

In exploring the robustness of code that has been mutated by genetic programming techniques, Schulte et al. define a neutral mutant as a mutated program that may be semantically different but still fulfills the program specification as defined by its test suite [23]. Using this definition of neutral mutant they show that on average more than 30% of mutations are neutral. Based off of their results they suggest that traditional definitions in mutational testing are too restrictive and reduce the acceptable number of variants that can be produced.

Genetic programming can also be used to ease the transfer of existing software implementations to hardware. Langdon and Harman demonstrate the idea of Genetic Interface Programming (GIP) [15]. GIP utilized genetic programming techniques to create interfaces as opposed to creating new programs or reengineering whole systems. By combining NVIDIA's CUDA framework and a genetic algorithm they were able to generate a parallel GPGU kernel that exhibited the same functionality as a highly optimized sequential Unix utility, *gzip*.

There is also active research being done on the security benefits of in-place modification such as the work done by Crane et al. to demonstrate a novel approach to defending against attacks that utilize dynamically bound functions such as Counterfeit Object Oriented Programming (COOP) attacks [6]. Utilizing a binary that has been instrumented during compilation as well as the insertion of software "booby traps" they are able to dynamically randomize and hide the contents of virtual tables at load time. Although their approach utilizes a specialized compiler they can apply it to a binary if enough information on the virtual tables can be recovered by static analysis. When tested on a benchmark program as well as a version of *Chromium* browser their implementation incurred an overhead of 8.4%.

2.2 Discovering Program Transforms

It is colloquially understood that for any sufficiently complex program, that there are an essentially unlimited number of semantically equivalent variants. While "whole program" semantics between two program variants may be identical, the instruction-by-instruction semantics can differ extensively. The result is that we can generate programs that use different instructions and that have wildly different instruction-level semantics while still meeting the same external criteria for functionality. This idea is particularly interesting when the program size is allowed to change, or when particular aspects of end-to-end program functionality (such as performance) are allowed to vary.

The core purpose of our research is to explore the space of program variants (or mutants) using empirical methods. To gain a better handle on possible techniques we have categorized methods for discovering program transforms into the following categories:

- **Formal:** Formal/Manual approaches define some set of useful (or potentially useful) rewrite rules such as those used in optimizing compilers. Most semantics-preserving transforms were discovered and defined manually using formal methods. Generally the discovery of a sophisticated transform requires somewhat arduous proofs. Once understood, even sophisticated transforms may essentially be directly written as rewrite rules and manipulated as mathematical operations. Formally proven transforms can be applied to these program sequences with very little risk to affecting program operation or robustness. Examples of manual approaches for transform discovery which are seen in real-world software include compiler optimizations and mathematically sound bit-twiddling hacks.
- **Generative:** Generative approaches describe the grammar of a language and a set of rewrite rules for generating semantically equivalent expressions. By traversing the world of grammatically correct statements we can use such an approach to generate expressions which are semantically equivalent, but syntactically distant. A generative approach may also be used to generate programs which are semantically similar using rewrite rules which define semantic similarity rather than strict equivalence.
- **Empirical:** An empirical approach to discovering transforms could take place using any of several techniques. Our approach uses observations of functionally equivalent programs, looks at program differences to discover candidate transforms, then performs empirical measurements of the success or failure of these transforms in new program contexts.
- **Combined Methods:** It isn't necessary to select a single method. Empirical observations of GA-derived mutations could be used as a starting point for formal verification to prove a relation between the original program segment and a mutated program segment. Formal methods can also be used to vet candidate transforms by demonstrating differences in semantics.

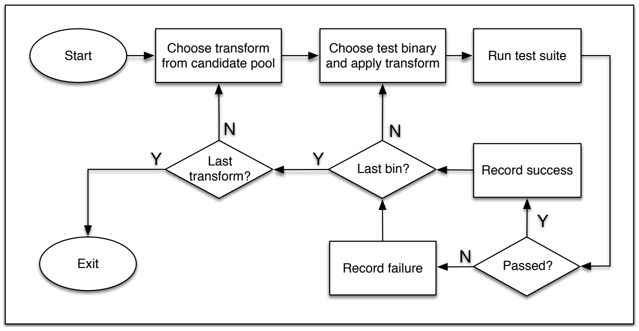

Figure 1: Rather than using formal methods, we use an end-to-end validation across a set of candidate programs to determine if our transform affects program functionality.

Our focus in this paper is on empirical methods for transform discovery and not on formal analysis or verification. Our empirical measurements do not represent any sort of proof of correctness, but are a rough estimate of the validity of each tested transform.

3 APPROACH

Our overall approach is detailed in Figure 1, and consists of the following steps:

(1) Mutate the target binary in place, producing thousands of variants.
(2) Test each variant for validity using a suite of unit tests.
(3) Discard variants which fail.
(4) Compare each passing variant to the original program.
(5) Extract candidate transforms meeting our *scope* criteria.
(6) For each candidate transform, apply it to a pool of test programs.
(7) For each test program, validate the transformed binary using available unit tests.
(8) Collect candidate transforms for further testing and manual analysis.

3.1 Program Mutation

Our initial pool of transforms was pulled from a group of variants that had been processed by a genetic algorithm to make as many changes to a program as possible while still passing that program's test suite. For our purposes, we mutated 16 programs within the set of GNU core utilities, coreutils: base64, basename, cat, chcon, chgrp, chmod, chown, chroot, comm, cp, csplit, cut, date, dd, df, dircolors. Program variants were generated using crossover with tournament selection as well as three possible mutation operations: copy, delete, and swap. For crossover we used two-point, where two offsets are chosen where both programs have instructions at both offsets. The resulting variant contains instructions from one of the programs up to the first offset, instructions from the second program up to the second offset, and then the rest of the instructions are from the first program. We compute the offsets for the mutation operations using the *objdump* utility. The algorithm terminates after a specified number of fitness evaluations and uses a population size of 512. We ran our algorithm on a 64 bit machine with 64 CPUs and 512 GB of RAM. Tests were assigned across 16 virtual machines with two GB of RAM and two CPU cores each. This was to allow reloading a virtual machine in the event that the virtual machine hangs due to faulty programs, such as writing too much data and filling up the disc. Runs for the genetic algorithm on a given program were terminated after 500,000 fitness evaluations and ranged from a couple of days to a week depending on the performance of the program's test suite.

All mutation operations were performed directly on the raw bytes of the binary image of the program as a executable file on disk. For a given transform and binary, we search the binary for the sequence of bytes which match the original instruction bytes and overwrite them with the bytes in the modified instruction bytes. This approach is very straightforward and allows for fast program mutations, but is likely to have limited applicability for programs which are obfuscated or have been modified for purposes of reverse engineering resistance (such as the use of misaligned instructions or self-modifying code).

Program	Bytes Changed	Percentage changed
base64	8694	52.266
basename	8495	69.166
cat	21013	71.898
chcon	27968	88.077
chgrp	25301	67.621
chmod	16959	54.171
chown	19745	58.303
chroot	13266	79.371
comm	9001	57.484
cp	28285	40.181
csplit	9499	41.820
cut	9885	54.644
date	13654	38.761
dd	14537	45.482
df	33181	60.715
dircolors	10784	68.314

Table 1: Change in executable code at 5x10^5 evals

Table 1 shows the percentage of executable code (in bytes) modified in resulting binaries generated by running the genetic algorithm on coreutil binaries for 500,000 evaluations. We see roughly a 50 percent change in the bytes in the executable section with a few notable outliers. For example, chcon had 88 percent of the executable code modified. Upon further investigation, chcon is used to change the SELinux security context for files. Since we were not using the SELinux module, some tests may have been skipped or failed to produce a fail condition due to early termination. In some other cases, programs worked correctly, but had minor issues such as unusual characters or words in the usage, or help, output. This further illustrates the dependence on test suites and the need for those with a good level of program coverage to reduce the risk of faulty transforms.

3.2 Candidate Transform Discovery

We identified three possible techniques for locating potential transforms uncovered by our genetic algorithm.

1. Compare a variant with the original (`diff`*)* This is the simplest and most straight forward method of candidate transform discovery. It is also the method that we used in this initial work. Given an original program and a function preserving variant it is possible to compare the machine code of the two programs and any differences in the machine code are candidate transforms. This method may also reveal potential transform chains. We consider a transform chain to be an initial transform that leaves the machine code in such a state where a second transform is able to be applied to the location that was changed by the initial transform. Care must be taken when dealing with a transform chain as it is possible to chain enough transforms to form a cycle. We did find transform chains in our candidate pool but none of the chains were cyclic and all the chains were broken after the exhaustive round of testing.

2. Take subsets of each candidate Given a safe transform that has been identified using method one we believe that for some transforms there may be a subset of other transforms within the larger change. These new transforms may be either novel transforms or previously discovered. In a cursory search of the initial pool of unsafe transforms we identified some larger transforms which contained smaller, already identified potential transforms. While none of these transforms were safe enough to make it through the first round of testing we believe that with a larger set of variants to examine or a set of variants that has been mutated with a different fitness function a safe version would emerge.

3. Take supersets with adjacent context (windowing of `diff` *context)* The last method we identified for discovering transforms involves including some of the unchanged instructions before and after the transform. For any given transform we can also consider supersets consisting of the original candidate as well as adjacent instructions selected using a sliding window of one to three or more instructions both before and after the candidate. Although we have not yet cataloged transforms using this approach, we believe this method would increase the number of safe transforms as it includes more of the program state, such as status flags from prior operations. This method would also drastically increases the search space.

3.3 Transform Vetting

For each of our candidate transforms, we are essentially hypothesizing that the transform is generalizable across other programs. We test this hypothesis by applying the transform to a selection of similar programs. We limit application of the transform to the *.text* section of the binary. This retains transforms which cannot be safely applied to modify data while also limiting the search space for useful transforms.

From the modified binaries, potential transforms were located by comparing an object dump of the original program to an object dump of a variant produced by the genetic modification process. Any differences in the machine code that are reported by `diff` are considered a potential transform. These potential transforms are then extracted and stored for use on other programs. To quickly weed out potential transforms that may be unsafe or machine/program specific we attempted to apply a single instance of each transform to other binaries in the coreutils set.

We define a *safe* transform according to the following definition:

> Given a program P, a binary transform T, and a program variant P' such that $P' = T(P)$ then T is considered a safe transform if P and P' are functionally equivalent, defined as both P and P' successfully passing a common test suite.

We believe that our testing approach is sufficient to ensure that a transform is *safe* in terms of "whole-program" validity. The specific semantics of the transform in question may violate program validity if assessed in terms of internal instruction-by-instruction semantics. Our approach is inspired by a wealth of prior work on program modification and mutational robustness as discussed in Section 2.1. The safety claim is also limited by the validity of the test suite being used. A test suite with poor coverage will result in transforms of limited validity. We chose to accept potentially risky transforms instead of profiling the program based on test suite coverage to obtain a large quantity of transforms. Our assumption is that these riskier transforms will be eliminated during the transform vetting process.

After the candidate transform was applied to a test program the modified test program was run through a series of test cases as shown in figure 1. We attempt to apply this pool of potential transforms to the 8.25 release of coreutils. We apply a single transform exhaustively to each member of the coreutil suite and then run the modified program through unit tests associated with the program. Any transform that is applied to a program which subsequently fails one of its unit tests is rejected as unsafe. Any candidate transform that is unable to be applied to any of coreutil programs is not rejected as unsafe but is still removed from the pool of candidate transforms. In other words, a candidate transform **must** be applied to at least one program in the coreutils suite and that modified program **must** pass all of the associated unit tests for the potential transform to be considered valid.

4 RESULTS

Our original GA population had a large number of transforms that worked without impact to test-based validation in at least one of the Linux coreutils. However, this number is a small subset of the total candidate transforms. As shown in table 1, some variants consisted of almost entirely modified instructions. Using test cases with better test coverage would reduce our overall number of candidates, but likely increase the percentage of validated transforms.

- Original candidates from GA population: 11,878
- Transforms working in >= 1 coreutils: 3445
- Transforms working in > 1 coreutils: 2041

Of the 3,445 validated transforms, 50 were found in all 107 of our test coreutils. On average, transforms were applied to 32 of the 107 test programs. We suspect these values are large because the programs were all compiled on the same system using the same compiler. We also suspect we can further eliminate transforms by attempting to apply transforms in combination, or by applying transforms to programs compiled with a compiler other than GCC.

4.1 Novel Transforms

While many of the transforms we found were simply deleting instructions by replacing them with an amount of NOPs that are equivalent to the instruction length there were a number of transforms that performed more interesting modifications.

The following transform performs exactly the same function, but using different instructions and different computations for the offset. This transform occurred in almost all of the coreutils, and appears to simply be performing cleanup of open files during program termination. This transform short circuits the call to `close_stdout` by simply loading the expected return address into `%rax` and leaving `stdout` open. The call to `close_stdout` often occurs in the epilogue of functions and while this operation is unlikely to be covered by common test suites, it is also unlikely to negatively affect program execution. So, while this mutation represents what appears to be a significant change in instruction semantics (dropping the `close_stdout` call entirely), it results in equivalent program behavior and identical system state shortly after program termination.

```
Original assembly:
 mov    %rsi,(%r12)
 je     404054 close_stdout+0x1154>

Transformed assembly:
 lea    0x39f4(%rip),%rax
 # 407a32 <version_etc_copyright+0xa12
 nop
 nop
```

Not all transforms modify the program in a positive manner. This transform appears to remove a part of the error handling from the modified program. We surmise that this transform was able to make it through the various test routines as it was in a section of code that wasn't touched by the test suites. We had two different versions of this transform successfully make it through the into the valid recursively applied pool, the one shown and another where the error handling is simply replaced with NOPs.

```
Original assembly:
 callq  401328 __errno_location@plt>

Transformed assembly:
 nop
 nop
 sub    %rbx,%rcx
```

4.2 Generalization

Our current test approach uses a very small subset of software samples that are all similar in construction and are run in a nearly identical environment. Our testing approach is unlikely to generalize to software which is on a different operating system, compiler, or processing architecture. Our long-term goal is to discover novel transforms that apply to *any* program. For this purpose we will likely need to expand the type and kind of mutations being performed. Our current cross-over approach borrows instructions solely from the program being mutated. We also only perform whole-instruction modification and instructions are not mutated in place. Many of the more interesting transforms (such as bit-twiddling hacks) are simply not part of our current search space.

To manage a large set of transforms, we used an SQLite database consisting of a single table. Columns in the table consisted of: original hex bytes, original assembly, transformed hex bytes, transformed assembly, counter for applications, counter for successes, and a validity flag. The hex bytes were required in order to apply the transformations in-place by searching for and overwriting portions of the binary file. The assembly is stored only for the benefit of analysis, providing a representation of the transform which is easier to read. The applications counter keeps track of how many times a transform has been applied. The successes counter keeps track of how many times a transform results in transformed programs which successfully pass their test suites. These counters were important in weeding out transforms earlier on in the process, as we only attempted transforms on the full test suite if the number of attempts and successes were equal. The last column was the validity flag. Transforms are validated against the larger test suite and are marked as: valid, invalid, or skipped. Valid transforms were those that resulting in programs which were able to successfully pass all tests. Invalid transforms resulted in programs which failed one or more of the coreutils test scripts.

5 CONCLUSIONS

The application of genetic programming to software repair, optimization, and general modification have proven incredibly fruitful. However, automated modification of programs and program evolution is still in need of computationally inexpensive and semantically expressive mutation operators.

Our work is partially inspired by the empirical techniques of analogous research in the sciences of biochemistry and evolutionary biology. Useful adaptations within cellular and metabolic machinery are naturally propagated as organisms adapt over evolutionary timescales. The scale and scope of modern software development efforts suggests that the software ecosystem is ripe for exploratory and empirical studies of this kind.

Our research in this area is still very early in its development and we would hesitate to make claims as to the utility of our approach. However, we feel strongly that systematic empirical exploration and transform validation are likely to play an important role in future research. In particular, there are a number of unexplored research areas that may prove fruitful.

5.1 Future Work

All of our test programs were compiled using GCC without optimizations. This raises the possibility that many of the transforms we've found are peculiarities only used by this particular compiler. More work needs to be done to see if transforms generated by one compiler are safe when applied to a binary generated by a different compiler.

We currently implement the mutation operations at the granularity of single instructions. We believe that if we allow mutation units to span a few instructions we may find additional transformation candidates of a more interesting nature.

Our current mutation operations are also essentially unbiased. However, there are many instances where performance or memory profiling could be used to guide mutations and thus decrease the number of epochs to achieve a particular fitness criteria. Combining profiling-based mutation with our approach is likely to result in the discovery of transforms with a higher-impact to program fitness.

Similarly, our approach could also be refined by performing dynamic tracing of software to determine which portions of a program are actually covered by a particular test suite. As such, program mutations could be constrained to portions of the program which are likely to have been exercised by the available tests. Program regions which are never executed through test cases could be excluded from mutation operations entirely. This technique would resolve many of the issues with untested features being excised or inadvertently modified.

It has also been suggested that in some instances, test cases may cover relatively unimportant features that have a significant performance impact. Removal or modifications of such features may have benefit but the challenges of gauging the relative importance of individual tests is not addressed with the current approach.

Another limitation of our approach is that we only perform mutations using machine code which is borrowed from the same program. We have discussed using large libraries of open-source software to "borrow" machine code samples for use in program modification. This expands the search space dramatically, but as we have seen even with our current approach, a large number of candidate transforms is quickly paired down to a small subset in our initial program variant validation.

We have also failed to address the human factors of program modification. Our technique deliberately avoids developer-guided modification with the goal of ad hoc mutation of software very late in the software lifecycle. Developers are unlikely to embrace the idea of arbitrary code mutations. However, it would be interesting to explore the use of developer-guided annotations or "interesting code here" hints. Such an approach may have the potential to improve the transform discovery process as well as modified program safety and performance.

Overall, while our current results and the extent of our experimentation has been very limited, we believe that the approach is sound and has potential to catalog and characterize a broad array of useful transforms.

6 ACKNOWLEDGMENTS

The authors would like thank the Office of Naval Research, Code 31 and SPAWAR Systems Center Pacific for their sponsorship of this research.

REFERENCES

[1] Bobby R Bruce, Justyna Petke, and Mark Harman. 2015. *Reducing Energy Consumption Using Genetic Improvement.* ACM, New York, New York, USA.
[2] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad. In *The 15th ACM conference on Computer and Communications Security.* ACM Press, New York, New York, USA, 27–38.

[3] J Lawrence Carter. 1977. A case study of a new code generation technique for compilers. *Commun. ACM* 20, 12 (Dec. 1977), 914–920.

[4] Brendan Cody-Kenny, Edgar Galván-López, and Stephen Barrett. 2015. *locoGP: Improving Performance by Genetic Programming Java Source Code.* ACM, New York, New York, USA.

[5] Jedidiah R Crandall, Roya Ensafi, Stephanie Forrest, Joshua Ladau, and Bilal Shebaro. 2009. The ecology of Malware. In *NSPW '08: Proceedings of the 2008 workshop on New security paradigms.* ACM Request Permissions, New York, New York, USA, 99.

[6] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *the 2015 IEEE Symposium on Security and Privacy (SP).* IEEE, 763–780.

[7] Stephen J Crane, Michael Franz, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, and Bjorn De Sutter. 2015. It's a TRaP. In *the 22nd ACM SIGSAC Conference.* ACM Press, New York, New York, USA, 243–255.

[8] Ang Cui and Salvatore J Stolfo. 2011. Symbiotes and defensive Mutualism: Moving Target Defense. In *Moving Target Defense.* Springer New York, New York, NY, 99–108.

[9] Sephanie Forrest, Westley Weimer, C Le Goues, and ThanhVu Nguyen. 2012. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on* (2012).

[10] Stephanie Forrest, Westley Weimer, ThanhVu Nguyen, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation.* ACM Request Permissions.

[11] Saemundur O Haraldsson, John R Woodward, Alexander EI Brownlee, and David Cairns. 2017. Exploring Fitness and Edit Distance of Mutated Python Programs. In *European Conference on Genetic Programming.* Springer, Cham, 19–34.

[12] Mark Harman, Yue Jia, William B Langdon, Justyna Petke, Iman Hemati Moghadam, Shin Yoo, and Fan Wu. 2014. *Genetic improvement for adaptive software engineering (keynote).* ACM.

[13] C A R Hoare. 1996. How did software get so reliable without proof? In *FME'96: Industrial Benefit and Advances in Formal Methods.* Springer Berlin Heidelberg, 1–17.

[14] Jason Landsborough, Stephen Harding, and Sunny Fugate. 2015. *Removing the Kitchen Sink from Software.* ACM, New York, New York, USA.

[15] W B Langdon and M Harman. 2010. Evolving a CUDA kernel from an nVidia template. In *Evolutionary Computation (CEC), 2010 IEEE Congress on.* IEEE, 1–8.

[16] W B Langdon and M Harman. 2013. Optimising Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation , PP (99) -. (2013)* 19, 1 (2013), 118–135.

[17] William B Langdon and Justyna Petke. 2017. Software is not fragile. In *First Complex Systems Digital Campus World E-Conference 2015.* Springer, Cham, 203–211.

[18] C Le Goues, Stephanie Forrest, Westley Weimer, and M Dewey-Vogt. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on.* IEEE, 3–13.

[19] A Newell and H A Simon. 1976. Computer science as empirical inquiry: Symbols and search. *Commun. ACM* (1976).

[20] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *Genetic Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 137–149.

[21] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. *An analysis of patch plausibility and correctness for generate-and-validate patch generation systems.* ACM, New York, New York, USA.

[22] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, Westley Weimer, Jonathan Dorn, and Jonathan Dorn. 2014. *Post-compiler software optimization for reducing energy.* Vol. 49. ACM.

[23] Eric Schulte, Zachary P Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (Sept. 2014).

[24] Eric M Schulte, Westley Weimer, and Stephanie Forrest. 2015. *Repairing COTS Router Firmware without Access to Source Code or Test Suites: A Case Study in Evolutionary Software Repair.* ACM, New York, New York, USA.

[25] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07.* ACM, New York, New York, USA, 552–561.

[26] Martin Velez, Dong Qiu, You Zhou, Earl T Barr, and Zhendong Su. 2015. A Study of "Wheat" and "Chaff" in Source Code. *arXiv.org* (Feb. 2015). arXiv:1502.01410v1

[27] Westley Weimer, Stephanie Forrest, and Eric Schulte. 2010. Automated program repair through the evolution of assembly code. In *ASE '10: Proceedings of the IEEE/ACM international conference on Automated software engineering.* ACM Request Permissions, New York, New York, USA, 313–316.

[28] David R White, Andrea Arcuri, and John A Clark. 2011. Evolutionary Improvement of Programs. *Evolutionary Computation, IEEE Transactions on* 15, 4 (2011), 515–538.

[29] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. *A study of equivalent and stubborn mutation operators using human analysis of equivalence.* ACM, New York, New York, USA.

[30] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on* 28, 2 (Feb. 2002), 183–200.