Noname manuscript No.
(will be inserted by the editor)

Software Mutational Robustness

Eric Schulte · Zachary P. Fry · Ethan Fast · Westley Weimer · Stephanie Forrest

the date of receipt and acceptance should be inserted later

Abstract {FIXME currently 290 words, must be below 250} Software is often viewed as brittle: even small mistakes can cause serious errors. We empirically investigate this intuition and find software to be surprisingly robust to certain standard code mutations. We present a formal definition of *software mutational robustness* which measures the preservation of software functionality after such mutations. We use the software's existing test suite to measure functionality, and adopt standard mutation operators from the field of genetic programming. Software mutational robustness is analogous to mutational robustness in biological systems. We investigate aspects of mutational robustness which may be general across both biological and engineered systems; mutational robustness permits the accrual of beneficial diversity, and facilitates evolutionary adaptation. These aspects of software mutational robustness may help explain many recent successes in the field of Software Engineering.

We empirically test the mutational robustness of 22 programs including 12 production software projects, the Siemens benchmark suite, and 4 programs specially constructed to permit exhaustive testing. Our results hold across all classes of programs, for mutations at both the source code and assembly instruction levels, across various programming languages, and are not entirely explained by test suite quality. We conclude that mutational robustness is an inherent property of software, and that neutral variants (i.e., those that pass the test suite) often fulfill the program's original purpose or specification.

We postulate that software mutational robustness is an opportunity to create useful diversity because software mutations often result in nontrivial alternative implementations. To demonstrate this, we generate and select diverse populations of neutral program variants. For a program with seven or more held-out latent bugs, we can, on average, automatically generate a population of seven neutral variants in which at least one variant is not vulnerable to one of the latent bugs.

Keywords mutational robustness · proactive diversity · genetic programming · mutation testing · equivalent mutant · reliability

1 Introduction

The modern software development environment is the product of over fifty years of continued use, appropriation and refinement by software developers. Those tools, design patterns, and codes which have survived this gauntlet of developer attention are those which are robust to software developer's edits, hacks and accidents.

This software edit robustness is similar to the mutational robustness of natural systems. Mutational robustness has been shown to be intimately linked with natural systems facility for unsupervised evolution and adaptation to new environments. We posit that software mutational robustness points to the potential for similarly powerful methods of unsupervised software enhancement and evolution.

The *environmental robustness* of software is an important issues in software engineering, especially as it relates to reliability, availability or dependability. Herein, we introduce the notion of the *mutational robustness* of software, we provide a formal definition of such and show it to be a pervasive property of modern software. Finally, we provide an example application which leverages the mutational robustness of extant software to increase its environmental robustness.

Given that as much as 90% of the cost of a typical software project occurs after development [1], automated methods of software maintenance and evolution are of great importance to software engineers. In addition to the potential practical applications of software mutational robustness, an analysis of mutational robustness in an ecosystem as rich and complex as modern software provides a unique opportunity to investigate those properties of robustness and evolution which are general across both natural and engineered systems.

1.1 Software Mutational Robustness

We define the "software mutational robustness" $MutRB(S,T,M)$ of the software S, test suite T and set of mutation operators M, as the fraction of all first-order mutants of S arising through application of $m \in M$ which pass T.

$$
\begin{aligned}
MutRB(S,T,M) = \frac{|S_M^T|}{|S_M|} \text{ where} \qquad (1)\\
S_M^T &= s | s \in S_M \vee s \text{ passes } T\\
S_M &= s_m | s_m \overset{\forall}{\leftarrow} m(S), \forall m \in M\\
S,T &= \text{the original software instance, and test suite}\\
M &= \text{the set of mutation operations}
\end{aligned}
$$

In this work we use a constant set of standard mutation operations M taken from the Genetic Programming community, we define these operators and our software representation in Section 3. The diverse benchmark suite of software instances used are described in Section 4.1 and the software's exiting test-suites (both high and low quality) are used unaltered. While software mutational robustness is a metric of the (S, T, M) triple, we find largely uniform mutational robustness scores with an average value of 36.75% and a minimum across all software instances of 21.2%. We thus propose that mutational robustness be seen as an inherent property of software.

Under this definition a mutated program variant is considered "neutral" if it passes T [1]. Thus "software mutational robustness" may be phrased as the portion of software mutants which are neutral. Neutral variants are equivalent to the original program with respect to the test suite, they may or may not be semantically equivalent (functionally identical) to the original program and they may or may not satisfy the specification (desired behavior) of the original program. Empirically we find that a large portion of neutral variants are both semantically distinct from the original program and still satisfy the original program's specification or intended behavior. We find that a program's test suite is often a sufficiently discriminatory metric of desired program behavior.

As an example of such a neutral mutation, consider this 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 to

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

changes the run-time behavior of the program without changing the output, giving an alternate implementation of the specification.

The mutation operators used in this work are described in detail in Section 3. Briefly, the three operators perform deletion, copy, and swap of program locations. These operators are commonly used in the genetic programming community where they have been shown to be sufficiently general and power to enable productive evolution. Additionally these operators are both plausible analogs of genetic operations on DNA, and edit operations which may be performed by a programmer. None of these operators require the creation of de-novo code.

While the mutational robustness of software is certainly highly dependent on the mutation operators used, and a competent programmer could likely craft operators capable of achieving any pre-set desired level of mutational robustness, we believe that the operators used in this paper are sufficiently

[1] not to be confused with the "equivalent mutants" of mutation testing, see Section 2.2.1

simple, powerful and general that they expose software mutational robustness as an inherent property of software.

We conjecture that high software mutational robustness helps explain the success of unsound program transformations. Examples of such transformations will be presented in Section 2.3.1 and may involve program modification at the source level (e.g., [2–5]) or changing values at run-time (e.g., [6]). Mutational robustness may help explain why program transformations, such as swapping two statements [5] or clamping an integer value [6], can produce acceptable program variants.

1.2 Contributions

The main contributions of the paper include:

- A formal definition of software mutational robustness and analysis of the semantic space (or the fitness landscape) of software. We discuss the relation between software functionality, software test suites and specifications.
- A methodology for the empirical measurement of software mutational robustness and a demonstration that mutational robustness is prevalent in off-the-shelf software. We find that 36.8% of our mutations produce variants that still successfully compile and pass all test cases regardless of test suite coverage, programming language, or representation level (source or assembly). We evaluate this claim using 22 programs involving over 150,000 lines of code and 23,151 tests.
- An application of software mutational robustness in the generation of populations of software variants with increase population-wide environmental robustness. We seeded bugs into 11 programs, and found that in 8 of these programs, populations of neutral program variants generated without knowledge of the seeded bug contained individuals which did not express the seeded bug. {FIXME We analyze these immune neutral variants for evidence of compensatory change or reversion of the seeded bug}. Across all programs, 17 out of 65 seeded bugs were repaired by at least one neutral variant. We also show that for a program seeded with at least 7 latent bugs, we can select a small population of neutral variants (4 ± 3) such that at least one variant is immune to at least one bug. Most real world programs ship with $\gg 7$ latent bugs.

In the remainder of the paper, we next review related work in both the Biology and Software Engineering communities in section 2. We then present our software representations and mutation operators in section 3. We present our experimental methodology and results in section 4. We present an example application of software mutational robustness in section 5. Finally, we analyze our results and discuss their implications in sections 6 and 7.

2 Background

2.1 Biology

The ability of biological organisms to maintain functionality across a wide range of environments and to adapt to new environments is unmatched by man made systems. Teasing out the intertwined mechanisms and evolutionary drivers which have produced the robustness and evolvability of biological systems is a major research topic. In this section we highlight portions of this field which inform our investigation of the mutational robustness of software systems.

A biological organism possesses both a genotype and a phenotype. The genotype is the static information which specifies the organism, and the phenotype is the acting organisms in the world. Each has a correlate type of robustness; *mutational robustness* and *environmental robustness* respectively. Mutational robustness is an organisms ability to maintain phenotypic traits in the face of internal genetic variation, and environmental robustness is the ability to maintain functionality across in a wide range of environments.

These two facets of robustness are closely related. Many of the causes of mutational robustness are also causes of environmental robustness [7]. In fact the pervasive mutational robustness of biological systems may be a by-product of their evolution for environmental robustness [8]. However mutational robustness has been shown to be beneficial in its own right, especially in its impact upon an organism's evolvability.

Many of the immediate single-mutation neighbors of a mutationally robust organism in the space of genetic variants will have fitness equal to that of the original organism. Connected sets of such neutral neighbors are called neutral spaces [9] and can span large swaths of an organisms fitness landscape [10]. Neutral spaces in fitness landscapes can contribute to an population's evolvability [11].

By drifting through neutral spaces a population may access new phenotypes located anywhere along the border of the neutral space, rather than being restricted to phenotypes which immediately border the initial population. Neutral spaces allow populations to increase diversity and to accumulate new genetic material. This buildup of material has been shown to be necessary for large evolutionary advancements [12–15].

In effect mutational robustness of an organism is a metric of the organism's fitness landscape. A number of metrics of fitness landscapes have been devised in an attempt to statistically characterize landscapes [16] and to directly measure the properties of a landscape which encourage evolution [17]. Our proposed metric of mutational robustness of software begins the work of applying such metrics to real-world software. We hope this research will both access the potential for improvement and open-ended development of extant software through evolutionary processes, and generalize the study of evolution from biological systems to engineered systems.

2.2 Software Engineering

2.2.1 Mutation Testing

The software engineering community has been studying program mutants for over 30 years under the mantle of "Mutation Testing"; however, their interpretation and use of program mutants has been very limited. In their landmark review of mutation testing Jia and Harmon introduce the field as follows [18].

> Mutation Testing is a fault-based testing technique which provides a testing criterion called the "mutation adequacy score". The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect [mutants] faults.

In mutation testing mutants are assumed to be faulty. Thus mutants which pass a program's test suite are taken as test suite failures and lower the test suite's "mutation adequacy score", a type of coverage metric. Mutation testing does recognize the possibility of "equivalent mutants" which are semantically identical to the original program (cf. the *Equivalent Mutant Problem* [19]). The mutation testing literature does not acknowledge the existence of the neutral mutants which are semantically different from the original program but which still satisfy the program's specification and thus must pass the test suite.

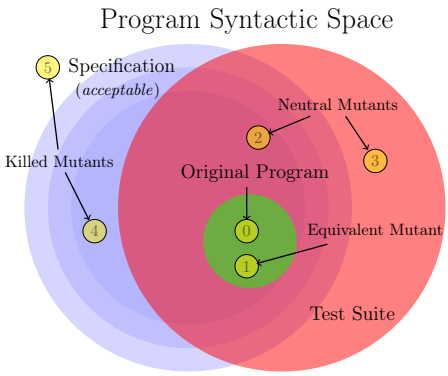


Fig. 1: Syntactic Space of a Program. The set of programs satisfying the program specification are shown in blue, the set of programs passing the program's test suite are shown in red, and the set of equivalent programs are shown in green. Three classes of mutants are shown and labeled. {FIXME dashed patterns for B/W printing}

Figure 1 shows the syntactic space surrounding a program. This is similar to a fitness landscape. The four possible classes of program mutants are labeled. This work will focus on the neutral mutants which have previously been ignored by the software engineering community. It is this novel biological interpretation of neutral mutants which separates this research from the field of mutation testing despite the many similarities in methodology and technique.

2.2.2 Fault Models

{FIXME Wes: do you think this is required background? It was mentioned as a glaring omission by a reviewer of our last paper, but I don't see much new germane material here.}

2.3 N-Version Programming

There has been considerable research on the use of automated diversity in security. An overview of current work in this area appears in a special issue of *IEEE Computer Security* devoted to IT Monocultures [20]. Common mechanisms for introducing diversity include Address Space Randomization [21,22] and Instruction Set Randomization [23,24], among many others. Diversity is introduced to reduce the risk of widely replicated attacks, by forcing the attacker to redesign the attack each time it is applied [25]. Our use of diversity is closer in spirit to n-variant systems [26], where the system runs the variants in parallel, giving each variant identical inputs, checking that they behave similarly before forwarding the output to the user.

Our proposed application also resembles n-version programming [27] where multiple independent, or quasi-independent, manually written implementations of critical programs reduce the risk of implementation errors going undetected. Our approach differs from n-version programming: Instead of relying on teams of programmers to generate full implementations, we use lightweight operators to automatically generate variants that are "nearby" the original. This addresses the cost issue identified in earlier studies [28]. As mentioned earlier, one limitation of n-version programming is the assumption that humans can generate programs that will fail independently; studies suggest that this assumption does not hold in practice [29]. Because we generate variants automatically, there is a better chance of achieving independence among the variations, either with the mutation operators we describe here or with others to be developed in the future.

2.3.1 Unsound Program Transformation

Traditionally, automated program transformation techniques (e.g., compiler optimizations) have been forbidden from altering the semantics (or behavior) of the original program. Such program transformations are "sound" because of this formally proven preservation of behavior. Recent work has experimented with "unsound" program transformations, which do not guarantee exact formal preservation of the original program, and are thus capable of more making more extensive changes to the original program.

Examples of unsound program transformations include *failure oblivious* computing [30], in which common memory errors such as out-of-bounds reads and writes are either ignored or automatically re-mapped from invalid memory addresses to arbitrary valid memory address. Such trade offs of robustness for

correctness are desire able in applications such as web servers where continued error-free operation is paramount.

Loop perforation [31] allows for the automatic trade-off of program accuracy for reduced running time or energy consumption. This techniques simply does not perform all of the computation specified in the original program by dropping a fraction of the executions of selected program loops.

Another important class of unsound program transformations are automated repair techniques. Such techniques take the same high-level approach: defining a notion of correct and incorrect program behavior (e.g., from test cases [6,5], implicit specifications [3], or explicit specifications [4]); generating a set of candidate repair transformations (e.g., at random [5], by constraint solving, or from an established set [3,6]), and validating the candidates produced through application of these transformations until a suitable repair is found.

Given that the process of automated software repair closely mirrors that of biological evolution, we propose that the inherent mutational robustness of software is an essential component of the success of these techniques; much as the biological mutational robustness has been shown essential to the evolution of biological organisms.

In the use of automated software repair, there are instances in which not only mutational robustness, but the resulting neutral spaces of software have been essential to the successful automated repair of buggy programs. {FIXME insert example of a repair which requires neutral mutation}.

3 Technical Approach

We study mutational robustness with respect to a set of mutation operators and a test suite, which assesses fitness by measuring the extent to which the software implementation adheres to its specification and satisfies its requirements.

For generality we consider two levels of program representation: abstract syntax trees (AST) based on high-level source, and assembly code (ASM). We use the CIL parsing toolkit [32] to generate ASTs. The ASM representation is the linear sequence of instructions taken directly from the compiled `.s` assembly code file produced by "gcc -O2 -S" on a 64-bit Intel platform, which is split on line breaks [33] , but with directives and other pseudo-operations kept unvaried. Our choice of one tree-based and one linear representation increases our confidence that the results do not depend on such representation details.

Given a source code or assembly language program, we consider three simple language-independent mutation operators: *copy*, *delete* and *swap*. Copy copies an AST statement-level subtree or assembly instruction and inserts it at a randomly chosen child position in a randomly chosen statement-level subtree or immediately after a randomly chosen instruction. Delete removes a randomly chosen statement-level AST subtree or assembly instruction. Swap exchanges two randomly chosen statement-level AST subtrees or assembly

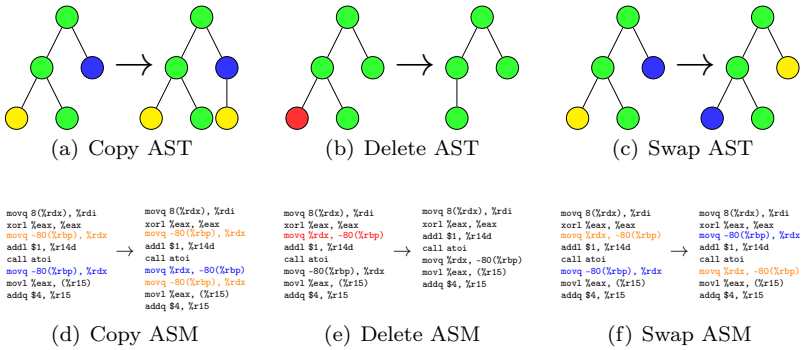


Fig. 2: Mutation operators: Copy, Delete, Swap.

instructions. Figure 2 illustrates these operators. Because AST mutations manipulate subtrees, a large amount of code might be inserted or deleted by a single mutation, depending on how high in the tree the mutation is applied. However, in all the experiments, mutations are restricted to modify only AST statements or ASM instructions that are actually visited by the test suite. Mutations to untested statements would likely be neutral under our metric, unfairly biasing the results towards overly high estimates of mutational robustness.

4 Experimentation

We report results for five experiments on the mutational robustness of programs in both representations (AST and ASM). We seek to establish: (1) the prevalence of mutational robustness in software, (2) that it is not fully explained by test quality, (3) a taxonomy of first-order neutral mutations, (4) that it is cumulative to multiple (i.e., higher-order) mutations, and (5) that it is present across multiple programming languages and paradigms.

4.1 Benchmark

We selected 22 programs for our experiments (Table 1). Twelve are off-the-shelf programs selected to demonstrate the applicability to real-world software. Six are taken from the Siemens Software-artifact Infrastructure Repository, created by Siemens Research [34] and later modified by Rothermel and Harold [35] until each statement, branch and def-use pair were covered by at least 30 test cases. The `space` test suite, which was generated by Vokolos [36] and later enhanced by Graves [37], covers every edge in the control flow graph with at least 30 tests. These programs are included for comparability to previous research and to demonstrate the robustness of programs with extremely high quality test suites. We include four simple sorting algorithms taken from

Program	Lines of Code		Test Suite		Mut. Robustness	
	ASM	C	# Tests	% Stmt.	AST	ASM
Sorting Algorithms						
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 Benchmark [34]						
grep	28776	10929	119	24.9	50.0	36.7
printtokens	2419	536	4130	81.7	21.2	25.8
schedule	922	412	2650	94.4	34.4	29.1
sed	17026	8059	360	42.0	33.0	25.6
space	18098	9126	13494	91.1	37.7	32.1
tcas	544	173	1608	96.2	33.5	25.9
Systems Programs						
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
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
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 or average	**664100**	**158571**	**23151**	**40.9**	**33.9** ±10.4	**39.6** ±21.5

Table 1: Benchmark programs with mutational robustness of first-order (one-step) mutations. "Lines of Code" columns report the size of the program in terms of lines of C source code and lines of compiled assembly code. The "Test Suite" columns show the size of the test suite both in terms of number of test cases and the percentage of all AST level statements in the program that are exercised by the test suite. The "Mut. Robustness" columns report the percentage of all first-order mutations that were neutral. The ± values in the bottom row indicate one standard deviation. For each program, at both the AST and ASM level we generated at least 200 unique variants using each of the three mutation operations (*copy*, *delete* and *swap*). Mutation operations were applied at locations chosen randomly from those visited by the test cases. For three programs (`bzip`, `ccrypt` and `oggenc`) we also evaluated on three independent alternate test suites.

`http://rosettacode.org` to demonstrate the robustness of programs with full statement, branch-level and assembly instruction test coverage.

Each program has an associated test suite. The tests either came with the program as part of its established regression test (e.g., Siemens, `potion`, `redis`, `jansson`) or were constructed manually (e.g., sorting algorithms, webservers). A number of our benchmarks implement invertible transformations

(e.g., compression, encryption, serialization, image manipulation), which form an implicit formal specification and simple testing [38]. Three of the programs (`bzip`, `ccrypt` and `oggenc`) are thus each evaluated on two independently constructed test suites. For `lighttpd` and `imagemagick`, we restricted mutations to `mod_fastcgi.c` and `convert.c` respectively, demonstrating application to modules as well as to whole systems.

4.2 Software Mutational Robustness

We first demonstrate that a variety of software programs exhibit significant mutational robustness under the mutation operators described in Section 3. In this experiment, we measure what percentage of random mutations leave the program's behavior unchanged on all of its test cases. We begin with a program that passes all of its test cases. For each of the mutation operators (copy, delete, and swap) we make at least 200 variant programs through a single application of the mutation operator to the original program. These are *first-order* or *one-step* mutations. All mutations are restricted to locations that are visited by at least one of the test cases. We then run the mutated program variant on its test suite and count the variant as neutral if it passes all of its tests.

Table 1 shows the results of this experiment on the benchmark programs. We wish to rule out trivial mutations, such as the insertion of dead code (e.g., statements that appear after a `return`) or the transposition of independent lines, that are visible in the source code but would produce equivalent assembly code. Since program equivalence is undecidable [19], we approximate this by compiling the AST using "gcc -O2 -S," which includes dead code elimination, SSA form, and instruction scheduling. Multiple source code variants that produce the same optimized assembly code (modulo label names and other directives) are counted only once in Table 1. Similarly, any two ASM-level mutations which produce the same executable are counted only once.

Although the results vary by program (e.g., `grep` is more robust than `printtokens`), the results show a remarkably high level of mutational robustness: Across all programs, operators, and representations (source or assembly) 36.8% of variants continue to pass all test cases with no systematic difference between the AST and ASM representation. In the next two subsections we ask to what extent these results arise from inadequate test suites (4.3) or from semantically equivalent mutations (4.4).

4.3 Relation to Test Suite Quality

The results in Table 1 could be an artifact of inadequate test suites, although we modify only code that is visited by the test suite. {LAUREN Similar to mutating parts of genome known to be involved in phenotype assayed.} {FIXME relate test suites to their biological analog} Figure 3 demonstrates that across

programs and test suites, test coverage has little correlation with mutational robustness (at most 17%). Due to potential overlaps between test cases, we report results for test suite coverage rather than test suite size. When test-suite coverage is plotted against mutational robustness, using the data from Table 1, there is no discernible pattern, either visually or statistically.[2]

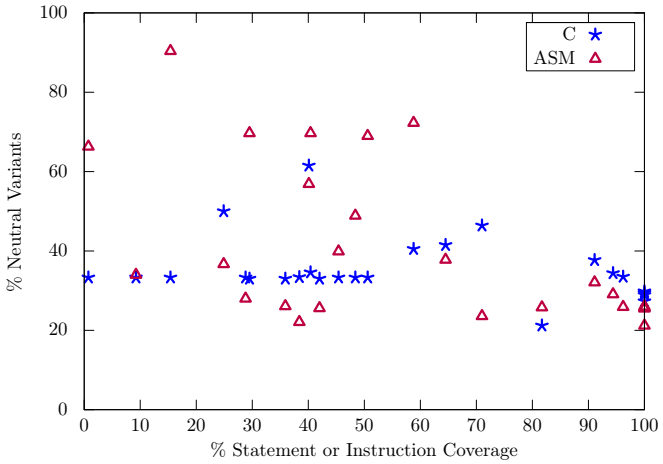


Fig. 3: Mutational Robustness by Test Suite coverage (replots data from Table I showing). The Pearson correlation coefficient between coverage and robustness is -0.41 ($\approx 17\%$).

One might expect mutational robustness to be fully inversely related to test suite coverage. This is not true in either limit. Even very high quality test suites (such as the Siemens benchmarks) and test suites with full statement, branch and assembly instruction coverage admit at least 20% of variants as neutral, while a trivial suite requiring only successful compilation and execution without crash admits only 84.8% of bubble-sort variants as neutral. Our results show that, in practice, for both real programs and comprehensively tested ones, mutational robustness can not be explained by test suite inadequacy. Thus, while some of the results in Table 1 can be attributed to test suite inadequacy, most cannot.

An even stronger claim can be made. The quick-sort example in Section 1 demonstrates that simple mutations can yield different fully correct implementations. Yin *et al.* provide another example in which a fully-formally verified implementation of AES encryption is changed based on what was "purely an implementation decision, [where] the specification did not impose any restrictions", yielding another formally verified implementation [39, p.61]. Additionally, estimates from the mutation testing community posit that be-

[2] Although we report Pearson correlations (assuming a linear relationship) for each data set, analysis using Bayes factor tests showed that the linear model is no more likely than generating the data randomly.

tween 5% and 15% of mutants are fully semantically equivalent to the original program [40]. These examples demonstrate that even with perfect validation, significant mutational robustness can still exist. This supports our claim that mutational robustness is inherent in software, rather than a function of inadequate validation, and it is at least partially explained by observing that for any program specification, there are infinitely many ways to implement it.

4.4 Taxonomy of Neutral Variants

To provide insight into the effects of first-order mutations on software we provide a taxonomy of the functional effects of 35 first order, AST, neutral variants of bubble sort. After manual review, all variants were confirmed to be valid implementations of the sorting specification. Variants were then categorized with respect to their operational differences from the original. The results are shown in Table 2.

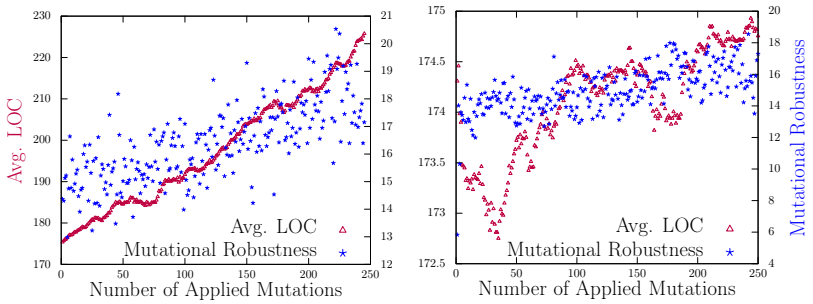
#	Functional Category	Frequency/35
1	Different whitespace in output	12
2	Inconsequential state change	10
3	Extra or redundant computation	6
4	Equivalent or redundant conditional guard	3
5	Switched to non-explicit return	2
6	Changed code is unreachable	1
7	Removed optimization	1

Table 2: Functional taxonomy of 35 neutral, first order, AST variants of bubble sort. Categorized by manual review.

These categories have different effects on program execution. Only categories 1 and 5 affect the externally observable behavior of the program by changing output and return values in ways not specified by the program specification. Categories 2, 3, 4 and 7 may affect the running time of the program. Category 2 includes the removal of unnecessary variable assignments, re-ordering non-interacting instructions and changing state which is later overwritten or never again read. Many changes such as 2, 3 and 4 produce programs that will likely be more robust to further manipulation by inserting redundant (occasionally diverse) control flow guards and variable assignments. Across most of these categories we find alternative implementations that are *not* semantically equivalent to the original but which do still conform to the program specification.

4.5 Cumulative Robustness

The previous experiments measured the percentage of first-order neutral mutations. In this experiment, we explore the effects of accumulating multiple



(a) Program size not controlled. (b) Program size controlled.

Fig. 4: Random walk in neutral landscape of Insertion Sort at ASM level. In Panel 4(a) the size of neutral variants is not controlled while in Panel 4(b) only variants which are less than or equal to the length of the original program (in ASM LOC) are counted as neutral. On the 32-bit machine used for this experiment `insertion-sort` compiles to 175 assembly LOC. Both the average size and the mutational robustness of mutant variants are given by the cumulative number of mutation operators applied.

neutral mutations in small assembly programs. We began with a working assembly implementation of insertion sort and generated 100 neutral variants using the procedure described earlier. We then iterated over the 100 variants, using them to generate 100 new neutral variants. This second set contains variants that are two mutations away from the original program. Figure 4 shows the result of repeating this process for 250 steps to produce a population of 100 neutral variants, each 250 mutations away from the original program.

As more mutations are applied to variants, it becomes easier to find additional neutral mutations. That is, mutational robustness increases along the random walk in the neutral landscape. This likely corresponds to movement away from the perimeter of the neutral plateau. Similar behavior is found in biological systems in which a weak evolutionary pressure for increased mutational robustness has been shown to exist in populations in a constant environment [41, Ch. 16].

The average size of the program also increases with distance from the original program Figure 4(a), suggesting that the program might be achieving robustness by "bloat"—adding in useless instructions. To control for bloat, Figure 4(b) show the results of an experiment in which we count as neutral only those individuals that are the same size or smaller (counted in assembly instructions) than the original. When this extra criterion is applied, mutational robustness continues to increase but the size periodically dips and rebounds. The dips are likely consolidation events, where additional instructions are discovered that can be eliminated. This shows that not only are the neutral spaces surrounding any given program implementation infinite, but they are easily traversable through iterative mutation, and contain sections with differing non-functional properties.

	C	C++	Haskell	OCaml	Avg.	Std.Dev.
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	39.5	28.6	18.6	27.9	
Std.Dev.	2.3	7.8	4.8	5.7	3.1	

Table 3: Mutational robustness of sorting algorithms at the assembly instruction level with 100% test suite coverage, for different algorithms and source language.

4.6 Multiple Languages

The source language and compilation process impose important regularities on the assembly representations of programs. In this section we investigate how mutational robustness varies across assembly code derived from different languages. We evaluated the mutational robustness of sorting algorithms compiled from five languages that span three programming paradigms (imperative, object-oriented, and functional). We focused on sorting algorithms because they are small and sufficiently well-understood to test exhaustively: Test suites were hand-crafted to cover all executable assembly instructions, branches, and corner cases.

The results shown in Table 3 demonstrate that mutational robustness exists across multiple programming languages and paradigms. It is striking that even for a small, exhaustively tested sorting program, 28% of the one-step mutations did not change the program's functionality.

OCaml produces slightly less robust assembly than other languages. Surprisingly OCaml variants have a higher rate of successful linking than their C counterparts (90.9% compared to 86.9% for bubble-sort), but the resulting executables are more likely to fail some or all of the test cases.

This experiment addresses the question of whether our results depend on idiosyncrasies of a particular language implementation or programming paradigm and supports the claim that robustness is an inherent property of all software.

5 Application

{FIXME re-write this whole section}

The previous sections demonstrated that mutational robustness is prevalent in software and not fully explained by language, algorithm, or test suite coverage. Section 4.5 suggests that mutations can be accumulated in the same program without loss of functionality. This suggests that it might be possible to make nontrivial deep changes to an algorithm or implementation within the neutral landscape. We now demonstrate a practical application of this idea:

Constructing diverse software variants to proactively detect and pinpoint unknown defects.

We hypothesize that only attackers and bugs depend on unspecified behavior, {LAUREN I don't understand this paragraph, sorry.} although we acknowledge that as a practical matter programmers sometimes rely on implementation details as a matter of convenience or performance. For example, the official C language specification leaves behavior undefined when programs index off the end of an array. This is why transformations that expand arrays or insert bounds checks are effective without interfering with normal users. Only attackers and bugs depend on any particular exploitable behavior, and artificial diversity leverages this observation [22]. We extend earlier work in this area by generating program variants that are distinct algorithmically but adhere to the same specification or test suite. These distinct variants can be used in an n-variant system [26], run concurrently, and flag potential bugs when there are discrepancies in observed behavior. When the variants contain algorithmic or implementation changes (rather than simple remappings, e.g., address space layout randomization or instruction set randomization), the approach is known as *implementation diversity* [42].

This technique may be viewed is a simple adaptation of traditional mutation testing. Wider adoption of the practice of mutation testing is limited due to the significant effort required for those mutants which do pass the test suite. Such mutants must be classified as equivalent or non-equivalent, and the latter further classified as a buggy version of the original or as superior to the original program (see *human oracle problem* [43]). Our proposed adaptation avoids these labor intensive steps by simply retaining a population of *possibly equivalent* programs. When a bug is encountered this retained population may be used to automatically identify the bug and suggest a repair. While a full manual review of all possibly equivalent programs (as required by mutation testing) is preferable, our approach is an improvement over normal industrial practice (making no use of software mutants) and is more easily enacted as it does not *require* exhaustive manual review.

5.1 Repairing Bugs

We first demonstrate that it is possible to construct variants in the neutral fitness landscape that can fix unknown bugs while retaining required functionality. We seeded each of the programs in Table 4 with five random defects following an established defect distribution [44, p.5] and fault taxonomy [45] (e.g., "missing conditional clause", "extra statement", "constant should have been variable", "wrong parameter"). The defects were seeded in advance and without regard to the mutation operators. For each defect we produced a held-out test to verify its presence or absence.

For each program, we then generated 5,000 one-step neutral variants by mutation, and noted if they also passed any of the held-out test cases for the seeded bugs. In practice, 5,000 neutral variants proved sufficient to generate a

bug-fixing variant for most programs, and if such a variant was generated at all, it was within the first 5,000 neutral variants. We tested this by searching up to 20,000 variants, which did not improve performance. Only variants that passed the original test suite were retained; the mutation process did *not* have access to the held-out test cases for the seeded bugs.

Table 4 shows the results. We say that a variant *fixes* an unknown bug if it passed all original test cases and the held-out test case associated with that bug. Such a variant also *pinpoints* the bug, because the `diff` between it and the original program can be used to locate the bug. {LAUREN Does this assume that the "fix" exactly reverts the original bug? Or are compensatory changes also possible?} Previous work demonstrated that software engineers take less time to address defect reports that are accompanied by such machine-generated patch-like information [46].

Although we did not conduct a systematic analysis of which bugs in Table 4 are fixed by neutral variants and which are not, we observe some common trends. The bugs fixed most easily were those that naturally mirror the mutation strategies employed by our technique. We found multiple examples of the following repairs in our experiment: deleting problematic statements or clauses, correcting incorrect values for constants, changing a relational operator (for instance $\leq$ to $<$), adding clauses or statements that test for extra conditions, changing a parameter value in a function call, etc. There was significant overlap between the types of bugs that were proactively repaired and those that were not, suggesting the need for more experimentation before drawing firm conclusions. However, one bug that was never fixed in our experiment involved an "incorrect function call." For our method to repair this, it would need to find the "correct" function elsewhere in the program with exactly the correct parameters (and avoid changing behavior on existing test cases). We leave for future work the problem of automatically determining how to resolve a discrepancy that is discovered among the variants, either with an automated repair or a new test case.

We hypothesize that the more latent bugs there are in a program, the more likely it is that at least one of them will be repaired through proactive diversity. This is relevant because most deployed programs have significantly more than five outstanding defects (e.g., [6]). To test the hypothesis, we seeded `potion` with 10 additional held-out defects; Figure 5 plots the number of distinct bugs fixed in 5,000 neutral variants as a function of the number of defects seeded, with correlation of 95%. Although preliminary, these results show how mutational robustness properties of programs might be used for practical advantage, for example, in the scenario outlined in the next subsection.

In their classic work on independence in multiversion programming, Knight and Leveson found that distinct teams given the same programming task produce implementations with correlated bugs [29]. That is, two independent teams are unlikely to produce two independent implementations. Our approach could potentially address part of the nonindependence hurdle by producing independent algorithmic changes. For example, if we seed 15 bugs in `potion` and then select at random ten of the variants that fix at least one bug, on average

Program	Fraction of Bugs Fixed	Bug Fixes
`bzip`	2/5	63
`imagemagick`	2/5	8
`jansson`	2/5	40
`leukocyte`	1/5	1
`lighttpd`	1/5	73
`nullhttpd`	1/5	7
`oggenc`	0/5	0
`potion`	2/5	14
`redis`	0/5	0
`tiff`	0/5	0
`vyquon`	1/5	1
average	1.0/5	18.8

Table 4: Five unique bugs were seeded in each subject program according to a defect distribution taken from a large open source project. Five thousand neutral variants were created for each program, without regard to the seeded bugs. Each variant passed all visible tests. The "Unique Bugs Fixed" column counts the number of seeded bugs fixed by at least one variant. The "Bug Fixes" column counts the number of variants that fixed at least one bug.

Diversity Metric Description	F	p
Maximum # of ASM lines changed in a function	23.70	0.01
# of *jump* or *goto* instructions changed ASM code	14.52	0.01
# of non-label ASM lines that differ	8.16	0.08
Average # of ASM lines changed per function	4.61	0.14
# of *if* or *else* lines changed in source code	2.04	0.29
# of ASM lines that differ	1.92	0.25
Percentage of functions altered	0.83	0.51
Ratio of ASM lines removed to ASM lines added	0.51	0.52

Table 5: Analysis of variants was performed over eight features used to compare a variant to its original program with the goal of identifying those features most correlated with the ability to fix withheld bugs. The F-value denotes the relative importance of the feature, and the p value denotes the significance level of F. All values are the averages of 10 randomized training data sets.

1.7 different bugs (rather than 1.0, if they were all dependent) are fixed by those ten.

5.2 Diversity Can Be Selected

We describe how to select a small set of first order neutral variants which are different from the original program in ways most likely to pinpoint a high percentage of the program's latent bugs. In an n-variant system, it is currently feasible to run a dozen copies of a program at a time on desktop hardware using techniques such as lightweight virtualization (e.g., [47]) but it is not reasonable to run 5,000.

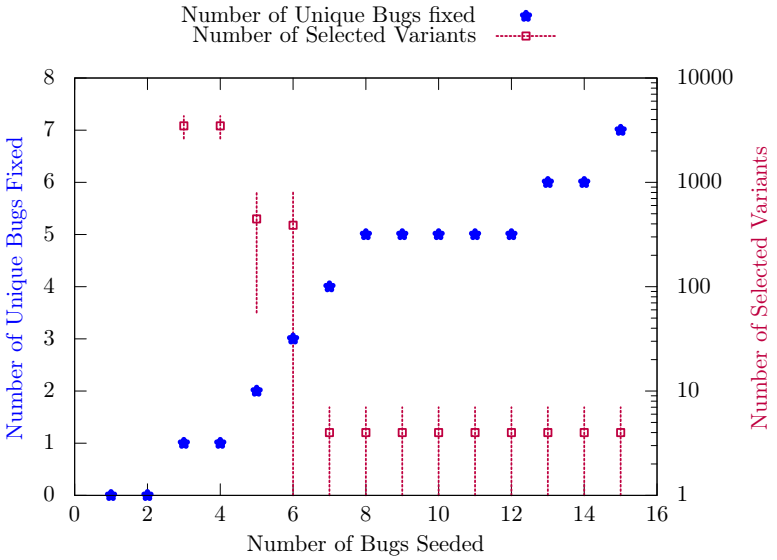


Fig. 5: Given a population of 5,000 program variants we show the total number of unique bugs fixed (blue) and the number of variants (selected using the features in Table 5) required before the first fix is found (red). Both are shown by the number of bugs seeded in the subject program (`potion`). The Pearson correlation coefficient between the number bugs seeded and fixed is 0.95.

As we desire algorithmic diversity, the selection metrics favor variants that have touched, changed or otherwise mutated as much of the program as possible, compared to the original. We measured eight distinct features (Table 5) and trained a linear regression model on `nullhttpd` and `bzip2` against their seeded bugs. We evaluated the selection technique by seeding between 1 and 15 bugs in `potion` and measuring the number of mutants that would have to be selected to fix any of those seeded bugs. Selection is performed by sorting mutants based on the model's output in descending order and taking a prefix of the list, thus including only the most "diverse" variants.

Figure 5 (points plotted in red) shows the results of this experiment. When as few as 7 bugs were present, our selection technique could identify a variant that served as a patch with a sample of only 4 distinct variants. As a baseline, sampling randomly at the same level would require an average of 14 (standard deviation of 16) variants to fix one of seeded bugs, suggesting that models based on simple diversity metrics and regression analysis have predictive power. {LAUREN I don't understand this experiment.} In practice, we find that the selection technique achieves high accuracy even when relatively few bugs exist.

We leave as future work the application of this technique to potentially much higher order neutral mutants (as constructed in Section 4.5). While

such variants would greatly increase diversity they would be less useful in pinpointing the source code implicated in buggy behavior.

To put this into perspective, for the first month after Mozilla 4.0 was released (March 11 through April 10, 2011), the project's Bugzilla database shows that an average of 77 new, non-duplicate bugs were reported per day. Given that this daily amount is more than 10 times the number of bugs seeded in our evaluation, we conclude that our technique would be at least as effective in similar real-world systems. Thus, our results are promising because they suggest that a small number of neutral variants run in parallel can pinpoint a relatively large number of bugs in practice.

6 Discussion

7 Conclusion

8 Acknowledgments

The authors gratefully acknowledge the support of the National Science Foundation (SHF-0905236), Air Force Office of Scientific Research (FA9550-07-1-0532,FA9550-10-1-0277), DARPA (P-1070-113237), DOE (DE-AC02-05CH11231) and the Santa Fe Institute.

References

1. R.C. Seacord, D. Plakosh, G.A. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices* (Addison-Wesley Longman Publishing Co., Inc., 2003)
2. V. Dallmeier, A. Zeller, B. Meyer, in *Automated Software Engineering* (2009), pp. 550–554
3. G. Jin, L. Song, W. Zhang, S. Lu, B. Liblit, in *Programming Language Design and Implementation* (2011)
4. Y. Wei, Y. Pei, C.A. Furia, L.S. Silva, S. Buchholz, B. Meyer, A. Zeller, in *International Symposium on Software Testing and Analysis* (2010), pp. 61–72
5. W. Weimer, T. Nguyen, C. Le Goues, S. Forrest, in *International Conference on Software Engineering* (2009), pp. 364–367
6. J.H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.F. Wong, Y. Zibin, M.D. Ernst, M. Rinard, in *Symposium on Operating Systems Principles* (2009)
7. R.E. Lenski, J. Barrick, C. Ofria, PLoS biology **4**(12), e428 (2006)
8. C. Meiklejohn, D. Hartl, Trends in Ecology & Evolution **17**(10), 468 (2002)
9. M. Kimura, *The neutral theory of molecular evolution* (Cambridge University Press, 1985)
10. P. Schuster, W. Fontana, P. Stadler, I. Hofacker, Proceedings: Biological Sciences pp. 279–284 (1994)
11. M. Huynen, P. Stadler, W. Fontana, Proceedings of the National Academy of Sciences **93**(1), 397 (1996)
12. S. Ciliberti, O. Martin, A. Wagner, Proceedings of the National Academy of Sciences **104**(34), 13591 (2007)
13. A. Wagner, Nature Reviews Genetics **9**(12), 965 (2008)
14. L.A. Meyers, F.D. Ancel, M. Lachmann, PLoS Comput Biol **1**(3), e32 (2005)

15. C. Ofria, W. Huang, E. Torng, Artificial life **14**(3), 255 (2008)
16. W. Hordijk, Evolutionary Computation **4**(4), 335 (1996)
17. T. Smith, P. Husbands, P. Layzell, M. O'Shea, Evolutionary Computation **10**(1), 1 (2002)
18. Y. Jia, M. Harman, IEEE Transactions on Software Engineering **99**(PrePrints) (2010). DOI http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62
19. T.A. Budd, D. Angluin, Acta Informatica **18** (1982)
20. IEEE security and privacy, special issue on IT monocultures. Vol. 7, No. 1 (2009)
21. S. Bhatkar, D. DuVarney, R. Sekar, in *USENIX Security Symposium* (2003)
22. S. Forrest, A. Somayaji, D.H. Ackley, in *Workshop on Hot Topics in Operating Systems* (1997), pp. 67–72
23. E.G. Barrantes, D.H. Ackley, T.S. Palmer, D. Stefanovic, D.D. Zovi, in *Computer and Communications Security* (2003), pp. 281–289
24. G. Kc, A. Keromytis, V. Prevelakis, in *Computer and Communications Security* (2003), pp. 272–280
25. D. Williams, W. Hu, J.W. Davidson, J. Hiser, J.C. Knight, A. Nguyen-Tuong, IEEE Security & Privacy **7**(1), 26 (2009)
26. B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, J. Hiser, in *USENIX Security Symposium* (2006)
27. B. Littlewood, D. Miller, IEEE Trans. Software Eng. **15**(12), 1596 (1989)
28. J.C. Knight, P. Ammann, in *IFIP Congress* (1989)
29. J.C. Knight, N.G. Leveson, IEEE Trans. Software Eng. **12**(1) (1986)
30. M.C. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, W.S. Beebee, in *Operating Systems Design and Implementation* (2004), pp. 303–316
31. S. Misailovic, D. Roy, M. Rinard, Static Analysis pp. 316–333 (2011)
32. G.C. Necula, S. McPeak, S.P. Rahul, W. Weimer, in *International Conference on Compiler Construction* (2002), pp. 213–228
33. E. Schulte, S. Forrest, W. Weimer, in *Automated Software Engineering* (2010), pp. 33–36
34. M. Hutchins, H. Foster, T. Goradia, T. Ostrand, in *International Conference on Software Engineering* (1994), pp. 191–200
35. G. Rothermel, M. Harrold, Transactions on Software Engineering **24**(6), 401 (1998)
36. F. Vokolos, P. Frankl, in *International Conference on Software Maintenance* (1998), pp. 44–53
37. T. Graves, M. Harrold, J. Kim, A. Porter, G. Rothermel, Transactions on Software Engineering and Methodology **10**(2), 184 (2001)
38. K.S.H.T. Wah, Software Testing, Verification and Reliability **10**(1), 3 (2000)
39. X. Yin, J.C. Knight, W. Weimer, in *International Conference on Dependable Systems and Networks* (2009), pp. 53–62
40. S. Segura, R. Hierons, D. Benavides, A. Ruiz-Cortés, Information and Software Technology (2011)
41. A. Wagner, *Robustness and Evolvability in Living Systems (Princeton Studies in Complexity)* (Princeton University Press, 2005)
42. C. Cowan, H. Hinton, C. Pu, J. Walpole, in *Proc. of the 23rd National Information Systems Security Conference (NISSC)* (2000)
43. E. Weyuker, The Computer Journal **25**(4), 465 (1982)
44. Z. Fry, W. Weimer, in *International Conference on Software Maintenance* (2010), pp. 1–10
45. J.C. Knight, P.E. Ammann, in *International Conference on Software Engineering* (1985)
46. W. Weimer, in *Generative Programming and Component Engineering* (2006), pp. 181–190
47. Y. Huang, A. Stavrou, A.K. Ghosh, S. Jajodia, in *Virtual Machine Security* (2008), pp. 19–28