Combining Bug Detection and Test Case Generation

Martin Kellogg
University of Washington
kelloggm@cs.washington.edu

Benjamin Floyd
University of Virginia
bef2cj@virginia.edu

Stephanie Forrest
University of New Mexico
forrest@cs.unm.edu

Westley Weimer
University of Virginia
weimer@cs.virginia.edu

***Abstract*—Detecting bugs is fundamental to building software. Static bug finding tools can assist in detecting bugs automatically, but they suffer from high false positive rates. Automatic test generation tools generate test inputs which can find bugs, but they suffer from an oracle problem. Both false positives and incorrect oracles cost precious developer time. This paper presents *N*-Prog, which combines the two approaches in a single system. *N*-Prog automatically identifies interesting, real input/output pairs in an input stream and presents them to the developer in an iterative fashion. For each such pair, the developer either classifies it as a bug (when the output is incorrect) or adds it to the regression test suite (when the output is correct). *N*-Prog uses random mutation to generate programs that are slightly different from the original but still pass the original test suite. It then selects input/output pairs whose input produces different output from the original on a on at least one of the programs—meaning that any test case created by *N*-Prog provably kills a mutant. Because every interaction between *N*-Prog and a developer either produces a new test case or a reproducible bug, *N*-Prog, by construction, has no false positives. Our experiments show that *N*-Prog builds regression test suites, detects defects, and finds test cases that have been removed from test suites. *N*-Prog combines bug detection and test case generation in a novel way that elimiates the time developers often spend examining false positives.**

***Keywords*-Mutation, mutational robustness, *N*-variant systems, *N*-Prog**

I. Introduction

Bugs are pervasive and expensive, and mature software projects ship with both known and unknown defects [2], [17]. Before fixing bugs, developers need evidence of their presence [60]—and acquiring this evidence earlier in the software lifecycle reduces each bug's cost [65]. To prevent defects in software shipped to customers, developers often use some combination of manual inspection, static analysis, and testing.

Each of these kinds of analysis has human costs. Manual inspection (i.e. code review) is expensive—it requires more than one developer to look at the code. In modern practice, code review is primarily used not to find defects, but for its other benefits—knowledge transfer, increased team awareness, and creation of alternative solutions to problems [4].

Static analysis tools [3], [5] can detect many classes of defects—even at industrial scale [10], [56]. Static analysis tools use a set of analyses designed to detect specific kinds of bugs, meaning that they are not fully general—they cannot detect a bug if no analysis tailored to that kind of bug has been written. In addition, static analysis tools suffer from false positives—their warnings may or may not correspond to real defects in the code. For instance, the developers of Coverity, an industrial-strength static analysis tool, report that they spend most of their engineering effort keeping false positive rates below 20–30% [10]. Each false positive wastes developer time investigating a non-issue, so eliminating this costly effort would improve developer productivity significantly. High false positive rates sometimes cause developers to abandon tools when they lose faith in the tool's correctness, causing defects which the tool could find to enter production. In practice, many developers avoid using static analysis tools to find bugs because of these issues [31].

Testing can, in theory, detect any bug, but in practice test suites are often incomplete. Because writing tests manually takes significant developer effort, researchers have developed automatic test generation tools [23], [51]. However, these tools suffer from an "oracle" problem—checking that outputs are correct requires that the tool know what the program under test should do [7]. Automatic tools can use implicit oracles—such as "segmentation faults should not occur"—to generate tests, or they can construct regression tests by using the program itself as the oracle. However, any oracle *not* examined by a human is only a heuristic. Such heuristics can be incorrect, and these cases require significant human effort to diagnose and fix [7].

We present a technique that combines bug finding and test suite generation in a way that addresses the limitations of each individual approach. The technique, implemented in a tool called *N*-Prog, presents its user with *alarms*, each of which either (1) is a new test case, including the correct output, or (2) indicates a bug in the program, along with some information that can aid in fault localization. Every alarm produced by *N*-Prog corresponds to one of these two situations, and thus, by construction, *N*-Prog produces no false positives. *N*-Prog replaces the "spurious warning" false positive of static analysis tools with useful new regression tests. Unlike a traditional static analysis, *N*-Prog is also fully general: it can potentially detect any bug that causes a change in the subject program's externally-visible behavior. The idea of removing false positives by combining two seemingly distinct processes is gaining in popularity (e.g., Jahangirova et al.'s approach to test input generation and oracle improvement [29]).

Like testing, *N*-Prog is a dynamic analysis. *N*-Prog creates a set of mutated variants of the subject program, all of which pass the same tests as the original. *N*-Prog then calls out to some method of input generation (such as a random input generator [51] or a cache of user data) to get an input stream. *N*-Prog then uses the mutated variants as a filter on the input stream: whenever an input causes different externally-

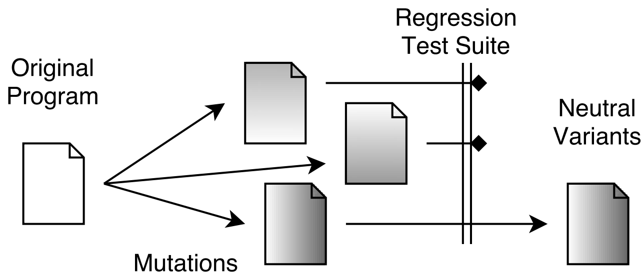

Fig. 1. *N*-Prog's variant generation process. Mutation operators are applied to the original program to create candidate variants. The candidate variants are run against the existing test suite (the double line in the figure). Neutral variants are those that pass.

visible behavior on these variants from the original, *N*-Prog alarms. Because *N*-Prog only alarms on inputs that cause divergent behavior, each input that alarms is either: (1) a new regression test that kills a mutant that the test suite could not differentiate from the original or (2) induces buggy behavior in the original program. In the former case, *N*-Prog is operating much like a mutation testing framework—but one which can improve the mutation score of the test suite by adding tests that kill mutants [30]. In the latter case, *N*-Prog is acting like a bug finding tool. A developer using *N*-Prog is continually improving their program: *N*-Prog automatically hunts for interesting inputs in an input stream, and only when it discovers one does the developer enter the loop—and every time the developer enters the loop, either a bug is discovered or a new regression test is added. As long as finding bugs and improving the regression test suite are valuable activities, *N*-Prog's results are always of value to the developer.

The primary contributions of this paper are:

- The *N*-Prog algorithm that combines bug detection and test case generation in a novel way to eliminate unnecessary human effort.
- A prototype implementation of the *N*-Prog algorithm for C programs.
- An evaluation of the *N*-Prog prototype, including a case study using a real webserver, a study of *N*-Prog's ability to detect defects and missing test cases, and a validation of an important algorithmic assumption. We find that *N*-Prog builds reasonable test suites, detects 32% of held-out bugs in our benchmarks, and that as test cases are removed from a test suite, *N*-Prog's ability to detect the missing test cases improves.

II. Algorithm

A. Overview

N-Prog acts as a filter on program inputs. It selects inputs that are guaranteed to be "interesting" to pass to developers—meaning that each input selected by *N*-Prog either elicits a bug or should be included in the test suite. *N*-Prog uses a combination of *mutation* and an *N*-variant system to achieve this goal. Here a mutation is a small random change to a program that modifies the program's abstract syntax tree [30]. A traditional *N*-variant system implements several different versions of the program—ideally with independent failure modes—and runs them in parallel [14]. When the variants *diverge*—execution of the same input produces observably different outputs—an *alarm* is raised. *N*-Prog creates its variants by mutating the source program, and when alarms are raised they are passed to developers. Every *N*-Prog alarm kills (i.e. rules out) some mutant, and the developer need only decide whether the original program was behaving correctly. If so, then the input and the original program's output form a test case—using the original program as the oracle, validated by the developer—that kills the mutant. If the original program is behaving incorrectly, then *N*-Prog has revealed a bug—and once the bug is repaired, the input can be paired with the output of the correct version to create a regression test that kills the original program.

N-Prog contains two major components: the variant generation process and the *N*-variant system. In Section II-B, we discuss the first component, and in Section II-C we discuss the second.

B. Variant Generation

N-Prog generates variants in two phases: first, candidate single-edit mutations are generated and tested to determine if applying them creates *neutral*[1] variants—variants whose behavior is indistinguishable from the original on the existing regression test suite. This phase is detailed in Section II-B1. Once *N*-Prog has generated a sufficient number of single-edit neutral variants, it moves to its composition phase. In this phase, *N*-Prog combines the single-edit mutations into multi-edit variants, each containing a configurable number of individually neutral mutants within a single variant. These higher-order variants combine the defect detection power of the original individual mutations (cf. [27]). This phase is discussed in Section II-B2. Pseudocode for both phases is shown in Figure 2.

1) Generating and validating single-edit mutations: This phase generates a set of neutral variants iteratively, as shown in Figure 2, lines 1–7. The algorithm makes a predefined number of *probes*, each of which involves a single mutation. This process is depicted in Figure 1. *N*-Prog uses a set of mutation operators to apply a single mutation to the subject program (the kinds of mutants our implementation generates are detailed in Section III-A).

N-Prog validates each variant with respect to the current version of the test suite. The current version of the test suite could be empty, however, in which case every mutant will be validated. The mutated program is compiled and executed against the test suite (`is_neutral` on line 4). If all tests pass, the mutation is considered neutral and added to the list of known neutral mutations (line 5); otherwise, it is discarded. Since the number of distinct first-order mutations for programs of even moderate size is large (e.g., typically well over 50,000

[1]We use the biologically-inspired term "neutral" following Schulte et al. [58], but "test-equivalent" and "sosie" are synonomous terms also appearing in the literature [9], [33].

Input: program of interest P.
Input: set of regression tests T for P.
Input: number of variants N for the final N-variant system.
Input: number of probes x, a budget for finding variants.
Input: number of probes y, a budget for finding multi-edit variants.
Input: maximum number of mutations k placed in a multi-edit variant.
Output: A set of up to n variants of P that each pass T.

```
1: neutral_variants ← Ø  //Generate mutants
2: repeat
3:    variant ← single_mutation(P)
4:    if is_neutral(variant, T) then
5:       neutral_variants ← neutral_variants ∪ {variant}
6:    x ← x − 1
7: until x ≤ 0
8: hom_variants ← Ø  //Begin composition
9: y′ ← y
10: while |hom_variants| < N do
11:    candidate ← choose_from(neutral_variants, k)
12:    if is_neutral(candidate, T) then
13:       hom_variants ← hom_variants ∪ {candidate}
14:       y′ ← y
15:    else
16:       y′ ← y′ − 1
17:       if y′ ≤ 0 then
18:          k ← ⌊k/2⌋
19:          if k ≤ 1  then
20:             return  hom_variants
21:          y′ ← y
22: return  hom_variants
```

Fig. 2. Pseudocode describing *N*-Prog's variant generation process. A set of single-edit mutations is generated, and then they are combined into multi-edit variants. Note that the input *T* (the test suite) can be the empty set.

for our benchmarks), it is not feasible to consider each one exhaustively. Instead, we make a fixed number of probes x at random (lines 6–7), and then only the neutral mutations are passed to the next phase of the algorithm for composition.

Note that each probe considers only one first-order (single-edit) mutation. These mutations are only a single step away from the original. Although higher-order mutations are likely to be more useful for *N*-variant systems, they are also less likely to be neutral. Previous work found that software is neutral to first-order mutations at a rate of about 30% [58]. Assuming independence, a random second-order, two-edit mutant would be expected to be neutral roughly 9% of the time. That is, a two-edit variant could potentially be twice as useful at detecting defects, but it is three times less likely to be neutral. We thus designed our algorithm to first generate a set of single-edit neutral mutations, and in a later step combine them into multi-edit variants. This strategy assumes that single-edit neutral mutants can be composed into multiple-edit neutral variants. We return to, and validate, this critical algorithmic assumption in Section IV-E.

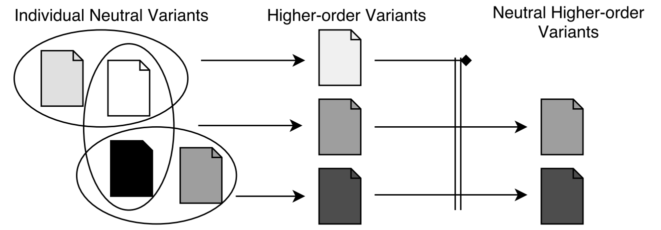

Fig. 3. *N*-Prog's composition process. To generate each higher-order variant, *N*-Prog takes the pool of neutral single-edit mutations and selects k mutations from it, then retests the combined variant for neutrality. The double line in the figure represents the regression test suite; the figure shows an *N*-Prog deployment with $N = 2$, $k = 2$, and $x \geq 4$.

2) Generating multi-edit variants: This phase composes non-interfering single-edit neutral mutants into multi-edit variants. Specifically, we aim to create N multi-edit variants, each comprised of k edits. This process is represented by the choose_from function on line 11 of Figure 2, and shown in Figure 3. Although the simplest approach would randomly select k edits, this occasionally produces interfering edits (i.e. edits that are neutral individually, but destructive when composed). *N*-Prog therefore applies a heuristic for ruling out some interfering edits (Section III-B).

When k non-interfering edits have been chosen or the permutation is exhausted (which can happen if fewer than k neutral variants with a single mutation were discovered in Section II-B1), the candidate variant is then tested once for neutrality (line 12). We found experimentally that our validity heuristic improves efficiency (reducing the number of non-neutral variants that are evaluated) and does not decrease the effectiveness of the generated variants.

It is not feasible to explore all permutations of a given set of neutral edits when considering candidate variants, so our algorithm has a composition search budget y. If y non-neutral candidate variants in a row are generated (line 16), we halve k (line 18) to increase the likelihood of generating a neutral variant. If k ever reaches 1, we terminate composition. Thus, sometimes variants contain fewer than k edits. This happens very rarely on reasonably sized programs programs (for instance, less than 1% of the benchmark scenarios in Section IV-B require k to be halved).

C. The N-variant system

The variants created by the composition phase (Section II-B2) are then deployed in an *N*-variant system. Figure 4 shows an example *N*-Prog internal *N*-variant system with three variants (each containing k mutations). Each input is copied $N + 1$ times, with one copy given to each variant and one to the original program. Next, each variant runs in parallel using its own copy of the input. Each variant's output is checked against the output of the original program by a monitor. A simple monitor (which we use in our experiments) is the Unix `diff` program—but a more complex monitor (such as one that monitors system calls [21]) could be used instead. If any monitor reports a divergence, the input that caused the

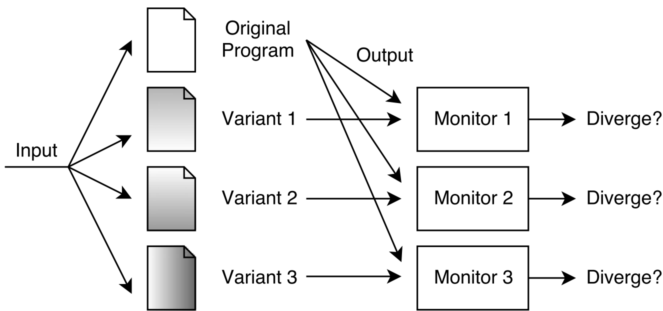

Fig. 4. *N*-Prog's internal *N*-variant system. Input, taken from the input stream, is copied to each variant, and the output of each is compared to the output of the original program. Only inputs that cause at least one variant to diverge are seen by developers.

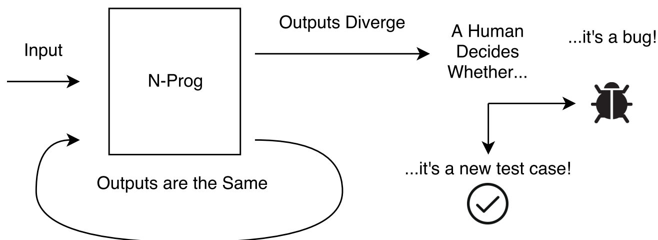

Fig. 5. Developer workflow when using *N*-Prog: Inputs are filtered, and only those that diverge between the original program and at least one variant are passed on to be reviewed by a human.

divergence and the original program's output are passed to the developer for inspection. Because a divergence indicates that this input exercises some behavior of the program that the test suite did not constrain, it must be interesting: either it is a new test case that can be added to the test suite or it indicates a bug in the original program.

N-Prog then uses this *N*-variant system as a filter on an input source; Figure 5 shows a high–level view of how *N*-Prog is used. Any input source can work—random input from a tool like Randoop [51], data collected from users, or any other well-formed input source. If every variant exhibits the same behavior for a given input as the original program, then *N*-Prog ignores that input and moves onto the next; if there is at least one diverging variant, *N*-Prog will issue an alarm. Note that while *N*-Prog uses machine time to consider non-diverging, non-interesting inputs, it discards them without showing them to a developer.

Once an alarm has been issued, there are two possible cases: either the original program is correct or the original program is incorrect. When it is incorrect, *N*-Prog has exposed a bug in the implementation, and the developers have a candidate patch that causes the program to exhibit different behavior (i.e. the mutated variant). The existence of a patch, even if it is incorrect, has been shown to aid in debugging [63]. When the original program is correct, then *N*-Prog has generated a test case, and the original program serves as its own oracle. Once this test case is added to the test suite, subsequent *N*-Prog variants will not alarm on that input. Generated tests such as these are of interest to developers because they make the test suite stronger—they kill mutants that the original test suite missed.

For each alarm, the only action required of the developers is examining the input/output pair (i.e., the given input and the original program's output) to determine whether the original program is behaving correctly. Usually, this requires much less effort than either writing a new test case from scratch or reproducing a bug, since examining an input/output pair is a part of either activity (cf. Section V).

III. Implementation

We implemented a prototype version of *N*-Prog that operates over C programs. This prototype is built on GenProg [40], and is available online via a git repository.[2] In this section we discuss design decisions that are specific to our implementation.

A. Mutation Operators

N-Prog's mutation operators are taken directly from GenProg [40]. We use only "delete" and "append" because they are atomic (modifying only one statement), and they can be composed to produce the same effect as other operators (i.e., "swap" and "replace"). We chose append over replace following the results of Baudry et al., who found append to be more effective than replace at creating neutral variants [9]. GenProg's mutation operators are coarse-grained, involving entire statements, which helps *N*-Prog—coarse mutations that cause undesirable behavior are likely to cause divergences [55]. When selecting mutations, certain ill-typed operations are ruled out in advance (e.g., mutations that would create programs that do not compile are not considered). In addition, we enable the optimizations proposed by Weimer et al. to GenProg [64], which reduce the search space of possible mutations by considering at most one representative from each equivalence class of program variants. By default, we restrict mutations to statements that are visited by the test suite as an artifact of using GenProg, which by default has this behavior. This choice is optional, and in our experience makes a negligible difference in *N*-Prog's success—but disabling it could only improve *N*-Prog's ability to find untested functionality.

B. Interference Heuristic

We use the following heuristic to decrease the probability of interfering mutations in one variant: two mutations are unlikely to interfere if they do not involve any of the same statements (meaning that, for instance, the same statement is not appended in multiple places in a single variant). Next, we generate a random permutation of the input set of neutral variants (`neutral_variants` in Figure 2). To produce a variant with k mutations, we take the first k non-interfering mutations in the permutation. In practice, we found that this heuristic strikes a good balance between including useful compositions and avoiding destructive compositions, while continuing to allow higher-order variants.

Another heuristic is our choice to sample with replacement, meaning that mutations that are placed in higher-order

[2]https://github.com/kelloggm/nprog-code

variants are not removed from the set of neutral mutations. Consequently, each mutation in the neutral set has equal probability of being placed in a given higher-order variant. We experimented with selection schemes that force or encourage all edits be to used (e.g., selecting without replacement), but found these methods produced a higher rate of non-neutral candidates without increasing bug detection rates, ultimately decreasing efficiency.

IV. Evaluation

To evaluate N-Prog, we designed a series of experiments that address its different capabilities in isolation. Taken together, the experiments show that N-Prog can consistently provide value to the developer. We report the results of experiments designed to answer the following five research questions:

- **RQ1:** How does N-Prog scale and does it apply to real-world workloads? (Section IV-A)
- **RQ2:** Can N-Prog detect held-out defects in a variety of programs? (Section IV-B)
- **RQ3:** What types of defects can N-Prog detect? (Section IV-C)
- **RQ4:** Can N-Prog variants detect missing tests? (Section IV-D)
- **RQ5:** Can individual neutral mutations be usefully combined into multi-edit variants? (Section IV-E)

Our evaluation features both real and toy programs, including some that are known to be tested exhaustively, some with real-world test suites, and both those with real and with seeded defects. When checking whether programs diverge, we always compare N-Prog output to the output that would be shown to the user (i.e. using `diff`). In general, we try to select parameters that result in reasonable variant generation times. N-Prog is relatively sensitive to both N, the number of variants, and k, the number of mutations in each variant: higher values of N and k lead to more alarms. Increasing each increases variant generation time, and increasing N also increases the time required to decide if each input is interesting. Increasing x also increases variant generation time and alarm rates, but less signficantly. Changing y has no impact on the results in more than 99% of cases. In the other cases, higher y results in higher variant generation time (cf. Section **??**).

A. *RQ1: How does N-Prog scale and does it apply to real-world workloads?*

1) Benchmarks and Experimental Setup: In this experiment, we measure the number of alarms that N-Prog raises with respect to an indicative webserver workload that is non-bug-inducing. We start with a single, simple test case for the webserver, so each alarm seen during the experiment corresponds to a new test case that should be added to the test suite. We apply N-Prog to `lighttpd 1.4.17`, a Web 2.0 webserver with a historical workload of 138,226 benign HTTP requests spanning 12,743 distinct client IP addresses [42, Sec. 6.2]. N-Prog is configured with $N = 8$ variants of $k = 30$ edits each and a search budget of $x = 400$, $y = 50$. These are subjected to the indicative workload. Each time an alarm is raised, the corresponding input is added to the test suite, with the original output as the oracle, and the eight variants are regenerated. Since N-Prog begins this experiment with a nearly empty test suite, we expect many alarms initially, as a test suite is generated to cover indicative behavior [30]. Since developer concerns over high alarm rates are an impediment for static analyses [31], ideally N-Prog will alarm only while building a good test suite.

Fig. 6. Alarms raised by N-Prog applied to a webserver with no test cases against an indicative workload of 138,266 requests (note log scale). Each alarm corresponds to a test case (input and oracle) discovered. The last alarm is raised at request 10,212.

2) Results: Figure 6 shows the results. N-Prog scales well, requiring the eight variants to be regenerated only 25 times over the course of 138,226 requests. Since each alarm represents both a CPU time cost (to generate mutations and compose them into multi-edit variants) as well as a potential human-time cost to determine if the alarm represents a test case or a defect, minimizing this number is important. In this experiment all inputs are known to be benign, so we do not consider the human judgment cost.

In the experiment the alarms occurred in the very early part of the run rather than being evenly distributed. The last alarm occurs 7% of the way through the input stream. This result is relevant to the real-world experience of using the tool—developers will not be bothered by many redundant alarms once a good test suite exists. Our experiment shows that as test cases are added to the test suite over time, N-Prog alarms only on novel inputs that exercise functionality that was not previously tested.

Since this experiment considered only non-bug-inducing inputs, we next evaluate N-Prog's ability to detect held-out defects.

B. *RQ2: Can N-Prog detect held-out defects in a variety of programs?*

1) Benchmarks and Experimental Setup: In this experiment, we measure the percentage of buggy inputs (i.e. inputs we, the experimenters, know are interesting) that N-Prog flags as interesting and passes to developers. To do this, we use buggy *scenarios*—versions of programs with a known bug and an input that illustrates the bug. First we generate N-Prog's

Program	LOC	Tests	Scen.	Alarm%
print_tokens	472	4,140	7	29%
print_tokens2	399	4,115	10	40%
replace	512	5,542	31	32%
schedule	292	2,650	9	11%
schedule2	301	2,710	10	30%
tcas	141	1,608	41	49%
tot_info	440	1,052	23	30%
potion	15K	220	15	27%
gzip	491K	12	5	80%
php	1,046K	8,471	62	21%
Total	1,593K	30,070	213	32%

TABLE I
Bug detection: Empirical results evaluating *N*-Prog's ability to detect bugs. Each scenario contains one held-out bug. The "alarm%" column reports in what percentage of the scenarios the held-out bug was detected by *N*-Prog for a given program.

N-variant system, using each program's existing regression suite (without the buggy input). The buggy input is then provided as the input stream; if *N*-Prog alarms, then *N*-Prog has detected the bug. If *N*-Prog does not alarm, we count it as a failure. For each scenario, we use a 25-variant system (i.e. $N = 25$) where each variant contains 30 individual mutations (i.e. $k = 30$), with search budgets of $x = 400$ and $y = 50$.

The benchmark set was selected from previously published work and includes both toy and real programs. The Siemens micro-benchmark programs (print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info) are each small C programs with a few hundred lines of code but many test cases—each has at least 30 test cases covering every line of source code [28]. Among the seven Siemens programs, there are 131 buggy scenarios. We also include potion, a larger, but not production, program that is an interpreter for a toy language [58]. potion has 15 scenarios with single bugs, and also 8 scenarios with multiple bugs. Both the Siemens programs and potion contain artificial, seeded defects. To test *N*-Prog on more natural bugs, we included two programs from the ManyBugs benchmark suite, which includes large, open-source programs with reported and repaired real bugs and developer-supplied test suites. We selected 67 scenarios from the gzip and php benchmark programs for which we were particularly confident of the test suites.

2) Results: The results of this experiment appear in Table I. Across all 213 buggy scenarios, *N*-Prog detected 68 (32%). At least one scenario was detected for each of the subject programs, and there were only two large outliers among the ten programs considered: gzip and schedule. On gzip, *N*-Prog was signficantly more successful than the rest of the programs, while on schedule it was significanlty less successful[3]. The 32% detection rate is conservative, because if *N*-Prog is given larger budgets (i.e. more variants in each system and more mutations per variant) it is possible it would be even more effective (but also more expensive to run). However, *N*-Prog's ability to detect about a third of defects is encouraging: if even a third of general software engineering defects can be detected automatically, while improving the subject program's test suite, a lot of developer effort can be saved.

[3]We suspect that *N*-Prog is so successful on gzip because gzip's regression suite is weak and gzip's output is highly sensitive to the kind of changes *N*-Prog makes.

Defect Count	Scenarios	*N*-Prog Detection %
1	15	27%
3	4	75%
5	3	67%
15	1	100%

TABLE II
A case study of *N*-Prog's ability to detect defects when multiple bugs are seeded into a single scenario, using the program potion. Note the sharp increase in detection rate when multiple bugs are introduced.

Defect Category	*N*-Prog	Cox et al.
Incorrect Behavior or Output	6/30	2/30
Security Vulnerability	1/4	4/4
Missing Functionality	1/11	0/11
Missing Input Validation	2/3	0/3
Spurious Warning	1/2	0/2
URL Parsing Error	1/1	0/1
File I/O Error	1/1	0/1
Fatal Error	2/9	0/9
Segfault	1/8	0/8
Bounds Checking	0/1	0/1
Memory Leak	0/1	0/1
Total	17/67	6/67

TABLE III
Comparison of the number of bugs actually detected by *N*-Prog and the number of bugs possibly detected by semantics-preserving transformations of the types used by Cox et al. [14]. Defects are broken down by type and taken from the ManyBugs programs gzip and php. Note that bugs classified as "security vulnerability" are also included in one other row depending on their cause.

Further, each scenario in Table I contains only one bug; in practice, real programs have many bugs. Scenarios with multiple extant bugs were only available for potion, but in those *N*-Prog performs significantly better than on single-bug scenarios, as shown in Table II. Over all eight multi-bug scenarios, *N*-Prog detected at least one defect in six; compare this with *N*-Prog's 27% detection rate on potion scenarios with only a single bug. Intuitively, this makes sense: if there are more bugs in the program, there are more program points where an *N*-Prog mutation can cause divergence that leads to bug detection. While this experiment only shows the results for a single program (and is therefore not generalizable), it suggests that deploying *N*-Prog in the real-world may result in higher detection rates than those presented in Table I.

C. *RQ3: What types of defects can* N*-Prog detect?*

Because *N*-Prog produces neutral variants, rather than equivalent mutants [30], it can theoretically produce divergence for many kinds of defects. We investigate what kinds of bugs *N*-Prog actually detects in practice. The ManyBugs programs (gzip and php) contain real, historical defects, in-

cluding scenarios that range from erroneous warning messages to security issues. In addition, these scenarios come equipped with a classification of each bug [41]. We therefore examined the classifications of the scenarios from the ManyBugs suite in detail. The majority of the considered bugs in these two programs were characterized as involving missing, extraneous or incorrect functionality.

Table III shows how many of each category of defect *N*-Prog was able to detect in the `gzip` and `php` scenarios in its middle column. Defects include at least one (the "Security Vulnerability" in `php`) marked "security critical." The table compares the *N*-Prog results to an earlier *N*-variant system (the right column in Table III) that uses semantics-preserving transformations [14]. When examining the bugs, we noted which could, in principle, be detected by the semantics-preserving *N*-variant system of Cox et al., which is designed to detect security defects at runtime by changing parts of the program's structure without changing its meaning (such as by modifying its address space or memory layout).

N-Prog is able to detect defects across a broad range of defect categories, demonstrating its generality. In principle, *N*-Prog can defect any defect that a change introduced by its mutation operators can impact—allowing it to apply to general software engineering defects, and not just defects of one particular type.

Three of the defects *N*-Prog detected were unusual: in each, an extraneous piece of information unique to the execution is printed as part of the output. For example, one bug in `php` (in the "Incorrect Behavior or Output" row of Table III) involves printing the value of a memory address, instead of printing the contents of a data structure. Such bugs could be detected by any neutral variant in an *N*-variant system. These three, including the one marked "security critical,"—which erroneously prints the names of temporary files—are among the cases that semantics-preserving transformations could have detected.

These results show that *N*-Prog can detect a wide variety of defect classes, including both security vulnerabilities and typical software engineering problems such as incorrect output.

D. ***RQ4:*** *Can* N*-Prog variants detect missing tests?*

This subsection investigates *N*-Prog's ability to detect missing tests with the following ablation experiment: we begin with a program and a minimal-size test suite with full statement coverage. We then remove, at random, a fixed percentage of the test suite; since the test suite is minimal, every test is necessary for full coverage. We then use *N*-Prog to generate variants based on the reduced test set, using the held-out tests as the input stream. We used the same settings for *N*-Prog as in Section IV-B (i.e. $N = 25$, $k = 30$, $x = 400$, and $y = 50$).

We report the percentage of generated variants that detect at least one missing test. The selected program, `tcas`, was chosen because it has associated minimized test suites [16]. We used all available minimal test suites (4 suites of 78 tests each). For each test suite, we remove 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90% of the test suite. At each percentage,

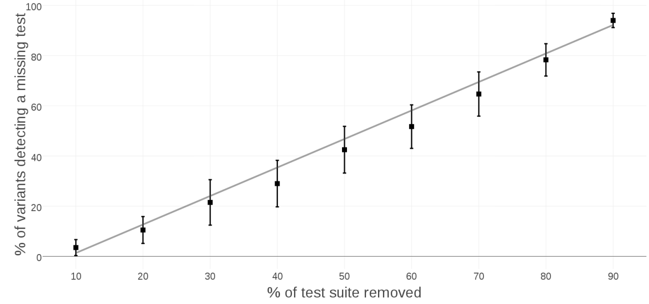

Fig. 7. Variant neutrality as a function of test suite availability using a minimal test suite. This graph reports the average percentage of `tcas` variants that detected at least one missing test case, when the variant was generated with X% of the test suite missing (horizontal axis). The trendline has a coefficent of determination of 0.989, and the error bars represent 95% confidence intervals.

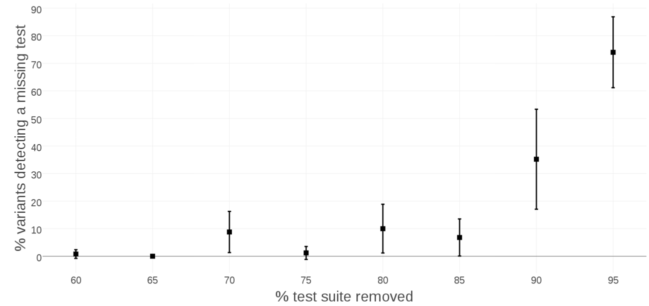

Fig. 8. Variant neutrality as a function of test suite availability using the full test suite. This graph reports the average percentage of `tcas` variants that detected at least one missing test case, when the variant was generated with X% of the test suite missing (horizontal axis)—note that this starts at 60%. When less than 60% of the test suite is removed, *N*-Prog does not alarm. The error bars represent 95% confidence intervals.

the experiment is repeated 10 times, with different random seeds. For each random seed, 25 variants are generated. In total, there are 250 variants generated for each percentage, 2,250 for each test suite, and 9,000 in total.

The results are shown in Figure 7. They indicate that *N*-Prog's ability to detect missing tests is linearly related to the number of test cases that have been removed—the best fit line (shown on the graph) has a coefficient of determination of 0.989, indicating a very strong linear relationship between the number of missing test cases and *N*-Prog's ability to detect them on `tcas`.

We repeated the experiment using `tcas`'s full test suite, which is very large (1,608 tests). Initial experiments showed that the results were very close to zero until most of the test suite was removed, so instead of sampling at 10% intervals, we sampled at 5% intervals, starting at 60%. The results are shown in Figure 8. When at least 15% of the test suite is present, *N*-Prog very rarely alarms; when the test suite is reduced further, *N*-Prog alarms often.

Between these two experiments, we observe that *N*-Prog is able to detect missing tests. When the existing suite is small

Composition	Useless	Useful	Non-Neutral
Useless / Useless	99.8%	0.0%	0.2%
Useful / Useless	0.0%	100.0%	0.0%
Useful / Useful	0.0%	100.0%	0.0%

TABLE IV
Results of 1,185 compositions of two single-edit, first-order mutation variants into one two-edit, second-order mutation variant. A useless variant passes tests but does not reveal defects, a useful variant passes tests and reveals a defect, and a non-neutral variant fails a test.

but has good coverage (Figure 7), *N*-Prog will detect more tests when more of the suite has been removed. *N*-Prog shows similar behavior with a large test suite, but only once the test suite has been reduced until there are few redundant tests.

In practice, these results indicate that as *N*-Prog is used and the missing tests it detects are added to the test suite, eventually almost all *N*-Prog alarms will correspond to the detection of a bug—a result that is consistent with our experience using the webserver and its indicative workload in Section IV-A.

*E. **RQ5:** Can single-edit neutral mutations be usefully combined into multi-edit variants?*

A key assumption of *N*-Prog is that multi-edit neutral variants can be generated efficiently from single-edit variants. To validate this assumption, we sampled and composed pairs of edits into two-edit variants. Neutral variants are those that pass all of the regression tests. However, some neutral variants will also behave identically to the original on the held-out bug-inducing input (we refer to those as *useless*), while others (referred to as *useful*) will behave differently (diverge) from the original on at least one bug-inducing input. Finally, a *non-neutral* variant fails at least one regression test. We consider the three possible pairings between two neutral single-edit variants: useless/useless, useful/useless, and useful/useful. The results of 1,185 such compositions are summarized in Table IV. These were selected from two benchmark scenarios: `tot_info` from the Siemens suite, and `digits`, a small program from the IntroClass suite of student-written programs [41]. This experiment suggests that compositions of useful variants are also likely to be useful ($p \ll 0.001$), and, as expected, composing two neutral variants produces a neutral combination in most cases ($p \ll 0.001$).

These empirical results give us confidence that *N*-Prog's assumption that atomic neutral mutations can often be combined effectively is true in practice, supporting the basic algorithmic design.

F. Threats to Validity

We now discuss some potential threats to the validity of these experiments. First, the benchmarks may not be indicative, threatening the generality of our results. We mitigate this threat by selecting benchmarks from previously published projects and by choosing programs with a broad spectrum of characteristics: sizes range from a few dozens of lines of code to more than a million; strong and weak test suites; and real and seeded bugs. All selected benchmarks are written in C—the *N*-Prog prototype currently handles only C, and programs written in other languages may have different behavior. However, prior work [57] showed that mutation operators similar to those we use can be applied successfully to x86, ARM, and ELF binaries for repair. This suggests that our approach may generalize to other program representations.

A threat to construct validity (cf. [19, Sec. 2]) is the use of test cases as a proxy for indicative workloads or developer inspection. We address this directly in Section IV-A, using an historical indicative workload of 138,226 HTTP requests. In other experiments, we measure "acceptable variant with respect to tests" when we would prefer to ascertain "acceptable variant in real life." Since developers rely on test cases in practice, however, we believe that this proxy is reasonable. When measuring the relationship between test suite quality and number of missing tests that are detected, we use the size of a minimized test suite as a proxy for incomplete or lower-quality test suites. Because we do not have access to indicative input data beyond the test suite for the programs in question, we follow the established practice of using tests. There is also a threat to generalizability from using only `tcas` to study how test suite adequacy affects *N*-Prog's ability to detect missing tests. Because we considered only minimized, coverage-adequate suites, the linear relationship we demonstrated may exist only under those conditions. We note, however, that real test suites can be viewed as coverage-adequate suites with some tests removed (i.e. the tests that have never been written were "removed" from the "perfect" suite).

G. Verifiability

Reproducible research is fundamental to the scientific process. Therefore, we have tried to make all of our results repeatable. Our prototype implementation of *N*-Prog is open-source[4]. The benchmarks we used in the experiments in this section are also available online[5], along with descriptions of how each experiment was carried out.

V. Limitations and Future Work

Generating multi-edit neutral variants can be expensive. The dominant cost of *N*-Prog is running the test suite of the subject program, which is necessary to validate the neutrality of each generated variant. Some programs have test suites that take only a few seconds to run, and for these programs, *N*-Prog can generate variants quickly—often in only a few minutes. However, other programs have significantly larger and longer-running test suites; for instance, `php`'s test suite takes over 30 minutes to run. Generating variants for `php` sometimes takes as long as several days on a single machine. We note, however, that variant generation is easily parallelizable, and multiple cores or cloud computing instances can generate variants independently. Thus, the latency of *N*-Prog remains low even when the cost in CPU-hours is high; in fact this

[4]https://github.com/kelloggm/nprog-code
[5]http://dijkstra.cs.virginia.edu/genprog/resources/nprog/icse-17-paper/

property allowed us to carry out the experiments reported here in a reasonable amount of time. For a discussion of the details of deploying large-scale, parallel experiments in the cloud, see Le Goues et al., whose model we followed [40]. *N*-Prog's compute cost is also dependent on its parameters—generating more variants take more time, so users could reduce the time it takes to generate variants by simply generating fewer variants. We believe that *N*-Prog's high cost in CPU hours is justified by its relatively lower cost in human time—a developer using *N*-Prog will never spend time looking at a spurious warning.

It is an empirical question how much effort is required for a developer to distinguish between the two different kinds of *N*-Prog alarms (i.e. buggy inputs vs. inputs that should be in the test suite). As a preliminary investigation, we conducted this decision process for several of our benchmark scenarios, chosen at random. In cases where we were familiar with the program's intended behavior, the decisions were made very quickly (under a minute on average). This is similar to results observed in the area of test-driven synthesis [6], [15], where users faced with input/output examples of candidate synthesized programs could quickly determine if the candidate output was correct. For larger or unfamiliar programs this process took longer (ranging from 10 minutes to an hour), which is consistent with previous reports that the effort required to address a defect increase with the time and organizational distance between the reporter and the developer [65]. However, our investigation was informal—one of the authors examined a few benchmark programs. A full investigation of this question would require a large-scale human study, with developers actually using *N*-Prog; we leave such a study as future work.

When applying *N*-Prog to a legacy system, the existing test suite has significant impact on *N*-Prog's performance: inadequate test suites will cause *N*-Prog to alarm more frequently for missing regression tests than bugs. Although formal specifications could help address the possibility of inadequate test suites (similar to previous work in automated program repair [49], [62]), we believe that *N*-Prog's current design is appropriate because it can help improve poor-quality test suites. In addition, *N*-Prog can be applied to the many existing programs which lack a formal specification. Nevertheless, an interesting avenue for future work would explore the use of specifications or contracts in conjuction with variant generation—the test suite currently constrains the behaviors of the mutants, but additional constraints could elicit more interesting behavior.

Different mutation operators might improve our results. We selected simple operators that sometimes make undesirable changes that can be validated by existing test suites [55]. Because that behavior is exactly what *N*-Prog needs to elicit divergence, we believe that the choice of these operators is appropriate. However, other mutation operators (e.g., from the mutation testing [30], [38] or automated program repair [35] literature) might be more effective. In particular, our mutation operators are very coarse-grained: they cannot make small changes in expressions (such as replacing a addition node with a subtraction node). These kind of changes are common in mutation testing, but we leave applying them in an *N*-Prog-like system as future work.

In this paper, we do not focus on the input stream that *N*-Prog requires. We noted (in Section II-C) that any well-formed input will work in theory, but because *N*-Prog acts as a filter on the input stream, it cannot raise an alarm on an input that is not in the stream. Thus, *N*-Prog's success depends in part on high-quality input. We consider this an orthogonal problem and note that test input generation is a mature, well-studied field [1]. Nevertheless, determining the best way to provide *N*-Prog with an input stream is an intriguing question. One possibility would be the use of shadow execution to find inputs that are likely to exercise the mutated parts of the code in each variant, using the techniques of Palikareva et al. [52]. We leave this exploration as future work.

VI. Related Work

Automatic Bug Finding. Both static and dynamic tools have been proposed to help find bugs in software. Static analysis tools—which reason about programs without running them—have a long history, ranging from early tools for C programs like Lint [32] to open-source tools like FindBugs [3] to modern, industrial-strength tools like Google's Tricorder [56] or Coverity [10]. All of these tools can find bugs—lots of them—but they also generate false positives, which waste developers' time. *N*-Prog eliminates the cost of these false positives. There have also been attempts to reduce false positive rates in static bug finding tools *post hoc*; Li et al.'s work on residual investigation is an example [43]. Dynamic program analysis, which does involve running the program under test, can also find bugs; tools like Daikon [18] or Valgrind [48] detect particular kinds of errors and also can produce false positives. Testing itself is a form of dynamic analysis, and so is *N*-Prog—both require running the subject program. However, every interaction between *N*-Prog and a developer leads to a useful result, and the types of defects *N*-Prog can detect are limited only by its mutation operators and the code on which they operate—unlike traditional static analysis tools.

Test Oracle Generation. Researchers have extensively studied the problem of automatically writing test cases. The so-called "oracle" problem limits the generality of this approach: only a human really knows what a system should do—any attempt to use that knowledge without involving a human must be an approximation [7]. Nevertheless, heuristics have been proposed that approximate human knowledge. Some tools use an implicit oracle (such as "the program should not crash") [25], while others use the program's current behavior as an implicit specification [22]. Still other techniques use an explicit, human-written specification as a guide [24]. Finally, some tools use other software artifacts, such as documentation, as oracles [26]. *N*-Prog avoids this problem by involving a human, who knows the system, when an oracle is required, but still drastically reduces the burden on the developer by only involving humans when inputs are known to be interesting.

Recently, Jahangirova et al. [29] proposed an approach for assessing and improving test oracles that is similar in spirit to *N*-Prog. They use search-based test case generation and mutation testing in tandem, building upon the EvoSuite tool [22]. Both approaches formally eliminate false alarms by exploiting the interplay between two techniques with a human in the loop, and for both one of the techniques used is mutation analysis. However, *N*-Prog combines mutation analysis and *N*-variant systems, while Jahangirova et al. combine search-based test input generation and mutation analysis. The approaches have the shared goal of eliminating false alarms and both use mutation analysis, but *N*-Prog focuses on *N*-variant systems and an extensive empirical evaluation, while the former work focuses on test input generation and formal descriptions of oracle improvement. In that regard the techniques are complementary, and we hypothesize that they could potentially be combined (with a shared mutation analysis informing both the *N*-variant deployment and test input generation).

Mutation Testing. Mutation testing creates mutants and uses them to determine the adequacy of a test suite by providing a *mutation adequacy score*, which indicates how effective a particular test suite is at "killing" mutants—if the mutant fails at least one test, it has been killed by the suite [30]. *N*-Prog could be viewed as an extension to the mutation testing paradigm. Where mutation testing is challenged by the so-called *equivalent mutant problem* [11]—some mutations cannot be killed by *any* test suite because they are semantically equivalent to the original—*N*-Prog takes advantage of neutral mutations, whether semantically equivalent or not, to protect against weaknesses in either the test suite or the program itself. This more ambitious approach can point to specific weaknesses in the test suite, much like mutation testing, but it can also pinpoint defects in the subject program.

Mutation in other Software Engineering Contexts. Mutation is used elsewhere in software engineering research. Mutation-based fault-localization tools use mutants to locate defects for which there is already evidence (i.e. a failing test case) [47], [53]. *N*-Prog finds defects, however, for which prior evidence does not exist while providing some localization benefits (in the form of the mutation that triggered the fault). Mutation-based generate-and-validate program repair tools, including GenProg [40], RSRepair [54], and Prophet [45] repair known defects by mutating program statements until a version of the program is produced that is "correct" by their measure. *N*-Prog does not explicitly attempt to repair defects, but rather pinpoints them. Many of these earlier tools rely on the program's regression test suite to validate fixes, as well, and *N*-Prog can be used to improve those regression suites to prevent "plausible but incorrect" mutants from being validated [44]. Schulte et al. [58] also used mutation to proactively repair defects before they had been detected, using neutral variants.

N-*variant Systems.* Chen and Avizienis first proposed a manual method for creating *N*-variant systems in which multiple teams work from the same specification [12]. However, in practice independent human teams do not typically produce independent implementations—even when they do not communicate [39]. Other approaches use semantics-preserving transformations to automatically generate a set of equivalent variants, which collectively can detect certain classes of security defects [14]. For example, each variant might use a different memory layout [59] or instruction set encoding [8], [34]. However, the restriction to semantically safe transformations limits the power of these techniques, unlike *N*-Prog. If the variants are all strictly equivalent to the original, many classes of defects cannot be detected (e.g., if a sorting program handles already-sorted input incorrectly, all of the equivalent variants will do the same, and no alarm will be raised).

Evolutionary Biology. *N*-Prog was inspired by a line of research in evolutionary biology that dates back to the 1960s. In nature, finite population sizes can lead to mutations becoming "fixed" (found in essentially all individuals of the population) even if they are not adaptive [36], a prediction made as early as 1930 by Fisher [20]. Because our algorithm can accumulate neutral mutations, it is an initial step towards incorporating the biological concept of genetic drift through "neutral spaces" [37] into software. In biological systems, this drift is posited as a necessary condition for non-obvious innovations [13], [46], [50], [61]. The success of *N*-Prog at detecting unknown defects suggests that this is potentially a fruitful area for future research.

VII. CONCLUSION

This paper presented *N*-Prog, a tool that combines bug detection with test case generation. *N*-Prog exploits weaknesses in each technique to augment the other: false positives become regression tests, and every time a human interacts with *N*-Prog, there is a positive outcome: either a bug is found or a test case which provably kills a mutant—complete with oracle—is written.

We demonstrated *N*-Prog's ability to detect held-out defects, achieving a success rate of 32% on a wide variety of general software engineering defects, as well as *N*-Prog's ability to detect test cases removed from existing test suites. We also demonstrated a plausible use case for *N*-Prog with a case study of `lighttpd`, demonstrating efficiency on 138,000 requests from an indicative workload.

N-Prog combines the activities of finding bugs and building regression test suites into a single system that eliminates the time developers spend evaluating false positives.

REFERENCES

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[2] J. Anvik, L. Hiew, and G. Murphy. Coping with an open bug repository. In *OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.

[3] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[4] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE Press, 2013.

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

[6] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *Programming Language Design and Implementation (PLDI)*, pages 218–228. ACM, 2015.

[7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering (TSE)*, 41(5):507–525, 2015.

[8] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanovic. Randomized instruction set emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[9] B. Baudry, S. Allier, and M. Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. *CoRR*, abs/1401.7635, 2014.

[10] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[11] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18, 1982.

[12] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *International Conference on Fault Tolerant Computing*, pages 3–9, 1978.

[13] S. Ciliberti, O. Martin, and A. Wagner. Innovation and robustness in complex regulatory gene networks. *Proceedings of the National Academy of Sciences*, 104(34):13591, 2007.

[14] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: a secretless framework for security through diversity. In *USENIX Security Symposium*, 2006.

[15] L. D'antoni, D. Kini, R. Alur, S. Gulwani, M. Viswanathan, and B. Hartmann. How can automatic feedback help students construct automata. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):9, 2015.

[16] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, Oct. 2005.

[17] S. M. Donadelli, Y. C. Zhu, and P. C. Rigby. Organizational volatility and post-release defects: A replication case study using data from google chrome. In *Mining Software Repositories*, pages 391–395, May 2015.

[18] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007.

[19] R. Feldt and A. Magazinius. Validity threats in empirical software engineering research - an initial survey. In *Software Engineering and Knowledge Engineering Conference*, pages 374–379, Redwood City, San Fransisco Bay, CA, USA, 2010.

[20] R. A. Fisher. *The genetical theory of natural selection*. Oxford, 1930.

[21] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[22] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

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

[24] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Foundations of Software Engineering (ESEC/FSE)*, pages 146–162. Springer, 1999.

[25] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium*. Internet Society, 2008.

[26] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis*, pages 213–224, 2016.

[27] M. Harman, Y. Jia, and W. B. Langdon. A manifesto for higher order mutation testing. In *Software Testing Verification and Validation Workshop*, pages 80–89, 2010.

[28] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow-and control flow-based test adequacy criteria. In *International Conference on Software Engineering*, pages 191–200, 1994.

[29] G. Jahangirova, D. Clark, M. Harman, and P. Tonella. Test oracle assessment and improvement. In *International Symposium on Software Testing and Analysis*, pages 247–258, 2016.

[30] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678, 2011.

[31] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[32] S. C. Johnson. *Lint, a C program checker*. Citeseer, 1977.

[33] R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 315–326. ACM, 2014.

[34] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Computer and Communications Security*, pages 272–280, 2003.

[35] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Sofware Engineering*, 2013.

[36] M. Kimura. Evolutionary rate at the molecular level. *Nature*, 217(5129):624, 1968.

[37] M. Kimura. *The neutral theory of molecular evolution*. Cambridge University Press, 1985.

[38] K. King and A. Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[39] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Trans. Software Eng.*, 12(1), 1986.

[40] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, 2012.

[41] C. Le Goues, N. Holtshulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. In *IEEE Transactions on Software Engineering*, 2015.

[42] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[43] K. Li, C. Reichenbach, C. Csallner, and Y. Smaragdakis. Residual investigation: predictive and precise bug detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):7, 2014.

[44] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *International Conference on Software Engineering (ICSE)*, pages 702–713. ACM, 2016.

[45] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Principles of Programming Languages*, pages 298–312. ACM, 2016.

[46] L. A. Meyers, F. D. Ancel, and M. Lachmann. Evolution of genetic potential. *PLoS Comput Biol*, 1(3):e32, 08 2005.

[47] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Software Testing, Verification and Validation (ICST)*, pages 153–162. IEEE, 2014.

[48] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, pages 89–100, 2007.

[49] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Sofware Engineering*, pages 772–781, 2013.

[50] C. Ofria, W. Huang, and E. Torng. On the gradual evolution of complexity and the sudden emergence of complex features. *Artificial life*, 14(3):255–263, 2008.

[51] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84. IEEE, 2007.

[52] H. Palikareva, T. Kuchta, and C. Cadar. Shadow of a doubt: testing for divergences between software versions. In *International Conference on Software Engineering*, pages 1181–1192. ACM, 2016.

[53] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[54] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *International Conference on Software Engineering*, 2014.

[55] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis*, pages 24–36, 2015.

[56] C. Sadowski, J. Van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*, volume 1, pages 598–608. IEEE, 2015.

[57] E. Schulte, J. DiLorenzo, S. Forrest, and W. Weimer. Automated repair of binary and assembly programs for cooperating embedded devices. In *Architectural Support for Programming Languages and Operating Systems*, 2013.

[58] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

[59] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Computer and Communications Security*, pages 298–307, 2004.

[60] Y. Tian, D. Lo, X. Xia, and C. Sun. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, 20(5):1354–1383, 2015.

[61] A. Wagner. Neutralism and selectionism: a network-based reconciliation. *Nature Reviews Genetics*, 9(12):965–974, 2008.

[62] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[63] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[64] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering*, pages 356–366, 2013.

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