Is the Cure Worse than the Disease? A Large-Scale Analysis of Overfitting in Automated Program Repair

Edward K. Smith☞ Earl T. Barr⋆ Claire Le Goues† Yuriy Brun☞
☞University of Massachusetts Amherst, MA, USA
⋆University College, London London, UK
†Carnegie Mellon University Pittsburgh, PA, USA
{tedks, brun}@cs.umass.edu, e.barr@ucl.ac.uk, clegoues@cs.cmu.edu

ABSTRACT

Recent research in automated program repair has shown promise for reducing the significant manual effort debugging requires. This paper addresses a deficit of earlier evaluations of automated repair techniques caused by repairing programs and evaluating generated patches' correctness using the same set of tests. Since tests are an imperfect metric of program correctness, evaluations of this type do not discriminate between correct patches, and patches that *overfit* the available tests and break untested but desired functionality. This paper evaluates two well-studied repair tools, GenProg and TSPRepair, on a 956-bug dataset, each with a human-written patch. By evaluating patches on tests independent from those used during repair, we find that the tools are unlikely to improve the proportion of independent tests passed, and that the quality of the patches is proportional to the coverage of the test suite used during repair. For programs with fewer bugs, the tools are *as likely to break tests as to fix them*. However, novice developers also overfit, and automated repair can, under the right conditions, outperform these developers. In addition to overfitting, we measure the effects of test suite coverage, test suite provenance, starting program quality, and the difference in quality between novice-developer-written and tool-generated patches when quality is assessed with an independent test suite from patch generation. We have released the 956-bug dataset to allow future evaluations of new repair tools.

1. INTRODUCTION

Automated program repair [3, 12, 16, 17, 24, 26, 29, 32, 33, 35, 39, 42, 43, 47, 54, 56, 57] holds great potential to reduce debugging costs and improve software quality. For example, GenProg quickly and cheaply generated patches for 55 out of 105 C bugs [29], while PAR showed comparable results on 119 Java bugs [26]. While some techniques validate patch correctness with respect to user-provided or inferred contracts [24, 42, 56], a larger proportion use test cases. The most common prior evaluations of automatic repair provide evidence of techniques' feasibility with respect to this test-case-based definition of patch correctness (e.g., [16, 31, 42, 56]).

However, in practice, a test suite is rarely exhaustive [48], and repair techniques must avoid breaking undertested functionality. When evaluations of a repair techniques use the same test cases or workloads to both construct patches and validate their correctness, they fail to measure whether or to what degree the repair technique breaks functionality. In our review of the literature, most of the prior evaluations of automated repair techniques that relied on test cases or workloads failed to evaluate patches independently of the test cases used to construct them. More recent work (e.g., [47, 60]) has begun to consider independent quality measures, though less extensively than we do here. And while some evaluations have used humans to independently measure repair acceptability [26] and maintainability [19], unlike our work, they did not directly evaluate patch correctness.

We refer to the repair techniques affected by this observation — ones that use test cases or workloads in patch construction — *generate and validate* (*G&V*) techniques. In this paper, we focus on *G&V* techniques. As we describe further in Section 2.2, *G&V* techniques are worth our investigation because they have broad applicability to mature, deployed, legacy software. Investigating other approaches, such as synthesis-based repair [24, 42, 56] techniques, is also of great value, but is outside the scope of this paper as it requires a different methodology and a focus on different kinds of input properties than is appropriate for *G&V* techniques.

Our contribution is a large-scale, controlled investigation of GenProg [31, 58] and TSPRepair [45, 46], both test-case-guided, search-based automatic program repair tools with freely available implementations that scale well to large programs. The evaluation identifies the circumstances under which these techniques break functionality despite producing patches that pass all test cases used during patch construction. We do this by using multiple test suites: one suite to construct the patch and another to evaluate it. To borrow from machine learning vocabulary, we use one test suite as "training" data to construct a patch, and another as "evaluation" or "held-out" data to evaluate the quality of the patch, checking if it breaks existing functionality. Patches that are overly specific to the training tests and fail to generalize to the held-out tests *overfit* to the training tests. Techniques that produce overfitting patches tend to fix certain program behavior while breaking other behavior.

The goals of our study are to (1) evaluate the quality of automated repair patches independently of their construction, and (2) measure the effects of properties of the input program and test suite on patch quality. Using *exhaustively testable* programs is thus a critical requirement for our study. Each program must be equipped with multiple independent exhaustive test suites so that patches can be evaluated for correctness independently from the suites used in patch construction. We also require a *large corpus* of programs to support a large-scale evaluation that results in statistically significant claims. We therefore produce a dataset for our evaluation by collecting 956 student-written programs with defects, submitted as homework in a freshman programming class, and all with student-written, bug-fixing patches. Each program specification is accompanied by two independent test suites: a *black-box* test suite written by the course instructor to the specification, and a *white-box* test suite constructed using the symbolic executor Klee [11]. We release this dataset to foster better evaluation of future automated repair tools: `http://repairbenchmarks.cs.umass.edu/IntroClass/`.

Previous work has used larger, real-world programs written by professional developers to evaluate techniques' scalability and applicability to complex program behavior (e.g., [29, 43]). However, these properties are not our focus. Although our use of small programs threatens the generalizability of our results, we make this tradeoff because we need many programs with multiple exhaustive test suites to evaluate the repair properties we are interested in. Understanding repair techniques at this scale increases understanding of repair techniques in general.

To the best of our knowledge, this is the first systematic effort to evaluate the correctness of automated repair with respect to fully independent measures. We measure overfitting and characterize repair quality along several previously unexplored dimensions, including test suite coverage, quality, and provenance. We also explicitly compare automatically generated and novice-developer-written patches with respect to functionality, as opposed to human judgments.

While our dataset is homogenous in that all our programs, bugs, and patches are short and written by novice programmers, it is rich in other ways, such as the availability of human-written patches, and a wide range of versions that fail multiple tests. The programs' small sizes, well-defined requirements, and numerous varied human implementations enable a comprehensive, controlled evaluation of how test suite coverage and provenance, patch minimization, and bug complexity affect repair quality. Automated repair is able to generate patches for many of the bugs in our dataset, providing sufficient data to draw statistically significant conclusions. Our study therefore increases the understanding of how, why, and under what circumstances search-based repair succeeds and fails through a large-scale experimental comparison, which would be dramatically more difficult (if not impossible) on large, complex programs. We find that:

- GenProg and TSPRepair are less likely to repair programs that fail more training tests.
- Both tools overfit to the training test suite used to guide patch construction. The resulting patches often break undertested functionality. Patch minimization does not reduce this effect.
- Test suite coverage is critically important to patch quality. Both tools produce patches that overfit more when given lower-coverage training suites.
- Both tools are more likely to break undertested functionality in programs that start with fewer defects. In these cases, the "fixed" program is often *worse* than the un-patched program.
- Both tools produce higher-quality patches when given human-generated requirements-based, black-box test suites than they do when given high-coverage, automatically generated, white-box test suites.
- Novice developers also overfit to provided test suites when fixing their own programs. In fact, they overfit more than the tools do when the tools use the black-box tests. However, the tools overfit more than the humans do when using the white-box tests.
- GenProg and TSPRepair can often generate multiple patches for the same bug. We find some evidence that combining multiple lower-quality patches can decrease overfitting, but the practical effect is quite small.

The rest of this paper is structured as follows. Section 2 summarizes automated repair. Section 3 describes our dataset. Section 4 discusses the results of a series of experiments measuring how the quality of inputs to automatic program repair affects the output patches. Section 5 presents a case study demonstrating overfitting. Finally, Section 6 acknowledges threats to the validity of our results, Section 7 places our work in the context of related research, and Section 8 summarizes our contributions.

2. AUTOMATED PROGRAM REPAIR

Automatic repair techniques can be classified broadly into two classes: (1) *Synthesis-based* techniques use constraints to build correct-by-construction patches via formal verification or inferred or programmer-provided contracts or specifications (e.g., [24, 42, 56] (2) *Generate-and-validate* (*G&V*) techniques create candidate patches (often via search-based software engineering [23]) and then validate them, typically through testing (e.g., [3, 12, 13, 16, 17, 26, 33, 35, 39, 43, 47, 54, 57, 58]. This paper focuses on *G&V* techniques. Section 2.1 defines this class of repair techniques, Section 2.2 explains the reasons we focus on it, and Section 2.3 discusses how we improve on prior evaluations.

2.1 Generate-and-validate program repair

G&V repair works by *generating* multiple candidate patches that might address a particular bug and then *validating* the candidates to determine if they constitute a repair. In practice, the most common form of validation is testing. A *G&V* approach's input is therefore a program and a set of test cases. The passing tests validate the correct, required behavior, and the failing tests identify the buggy behavior to be repaired. *G&V* approaches differ in how they choose which locations to modify, which modifications are permitted, and how the candidates are evaluated.

Of all existing *G&V* techniques, and to the best of our knowledge, GenProg [29, 58], TSPRepair [45], and AE [57] are the only publicly available repair tools that both repair programs written in C and target general-purpose bugs (as opposed to focusing on one domain of bugs, such as concurrency or integer overflow). In this paper, we use GenProg and TSPRepair as exemplars of *G&V* program repair. Unlike GenProg and TSPRepair, AE is deterministic, and so much of our experimental methodology does not apply. However, we do find that AE similarly overfits to the input tests (Section 4.2).

GenProg [29, 58] uses a genetic programming heuristic [28] to search the space of candidate repairs. Given a buggy program and a set of tests, GenProg generates a population of random patches. The fitness of each patch is computed by applying it to the input program and running the result on the input test cases; a weighted sum of the count of passed tests informs a random selection of a subset of the population to propagate into the next iteration. These patch candidates are recombined and mutated to form new candidates until either a candidate causes the input program to pass all tests, or a preset time or resource limit is reached. Because genetic programming is a random search technique, GenProg is typically run multiple times on different random seeds to repair a bug.

TSPRepair [45] uses random search instead of the genetic programming approach to traverse the search space of candidate solutions. Instead of running an entire test suite for every patch, TSPRepair uses heuristics to select the most informative test cases first, and stops running the suite once a test fails. TSPRepair limits its patches to a single edit. TSPRepair is more efficient than GenProg in terms of time and test case evaluations [45]. The same approach is also called RSRepair [46], and we refer to the original algorithm name in this paper.

There are three key hurdles that *G&V* must overcome to find patches [57]. First, there are many places in the buggy program that may be changed. The set of program locations that may be changed and the probability than any one of them is changed at a given time describes the *fault space* of a particular program repair problem. GenProg and TSPRepair tackle this challenge by using existing fault localization techniques to identify good repair candidates. Second, there are many ways to change potentially faulty code in an attempt to fix it. This describes the *fix space* of a particular program repair problem. GenProg and TSPRepair tackle this challenge using the observation that programs are often repetitive [6, 20] and logic implemented with a bug in one place is likely to be implemented correctly elsewhere in the same program. GenProg and TSPRepair therefore limit the code changes to deleting constructs and copying constructs from elsewhere in the same program. Finally, as a challenge that applies to GenProg in particular, genetic programming is known to lead to bloat, in which solutions contain more code than necessary [22]. GenProg minimizes code bloat post-facto; prior work has claimed that minimization reduces patches overfitting to the training tests [31]. TSPRepair only attempts single-edit patches, and thus does not further minimize successful patches.

GenProg and TSPRepair share sufficient common features to allow consistent empirical and theoretical comparisons. For example, in our experiments, we use the same fault localization strategy and fix space weighting schemes for both. This allows us to focus on particular experimental concerns and mitigates the threat that unrelated differences between the algorithms confound the results. However, the algorithms vary both in the way they traverse the search space and in the way they evaluate candidate patches, and thus we expect our findings to generalize to other *G&V* techniques, especially in light of recent successes in modeling and characterizing the similarities in *G&V* approaches [57].

2.2 Our focus on G&V

Our evaluation focuses on *G&V* approaches for two reasons:

First, while both synthesis-based and *G&V* techniques share high-level goals, they work best in different settings, and have different limitations and challenges. For example, the performance of synthesis-based repair relates strongly to the power of the underlying proof system, which is typically irrelevant to *G&V* repair.

Second, *G&V* is particularly promising for deployed, legacy software, because it typically does not require that the program be written in a novel language or include special annotations or specifications. As examples, Clearview, GenProg, Par, and Debroy and Wong have successfully fixed bugs in legacy software. Although new projects appear to be increasingly adopting contracts [18], their penetration into existing systems and languages remains limited. Few maintained contract implementations exist for widely-used languages such as C. As an example, as of March 2014, in the Debian main repository, only 43 packages depended on `Zope.Interfaces` (by far the most popular Python, contract-specific library in Debian) out of a total of 4,685 Python-related packages. For Ubuntu, 144 out of 5,594 Python-related packages depended on `Zope.Interfaces`. Synthesis-based techniques show great promise for new or safety-critical systems written in suitable languages, and adequately enriched with specifications. However, the significance of defects *in existing software* demands that research attention be paid at least in part to techniques that address software quality in existing systems written in legacy languages. Since legacy codebases often are often idiosyncratic to the point of not adhering to the specifications of their host language [8], it might not be possible even to add contracts to such projects.

2.3 Prior program repair evaluations

There have been several prior evaluations of *G&V* repair techniques. Most such evaluations demonstrate by construction that the technique is feasible and sufficiently efficient in practice [16, 31, 33, 35, 39, 42, 43, 54, 56, 58], some show that the resulting patches withstand red team attacks [43], some illustrate with a small number of examples that *G&V*-generated patches for security vulnerabilities protect against exploits and fuzzed variants of those exploits on typical user workloads [31], and some consider the fraction of a set of bugs their technique can repair [26, 29, 39]. These evaluations have demonstrated that *G&V* can repair a moderate number of bugs in medium-sized programs, as well as evaluated the monetary and time costs of automatic repair [29], the relationship between operator choices and test execution parameters and success [30, 57], and human-rated patch acceptability [26] and maintainability [19]. However, these evaluations have not used a metric of correctness independent of patch construction.

Our evaluation measures patch correctness independently of patch construction. We empirically examine how test suite coverage and provenance, number of test failures, and patch minimization affect repair effectiveness, defined by both success and functional correctness. We perform these experiments using a much larger set of bugs than ever before, designed to permit controlled evaluations that isolate particular features of the inputs, such that we can examine their effects on automatic repair in a statistically significant way.

Stressing the importance of this work, concurrent research is starting to evaluate repair techniques in terms of overfitting [47, 54]. Tan et al. [54] evaluate the degree to which *relifix* and GenProg introduce regression errors. Their evaluation is a step toward the independent correctness evaluation we advocate here, where we use independent test suites to measure patch quality. By contrast, those experiments use the subset of the original test suite that does not execute any of the lines associated with the bug under repair, ignoring specifically regressions a patch is most likely to introduce. Another concurrent evaluation is finding that poor-quality test suites result in patches that overfit to those suites [47]. Our evaluation goes further, demonstrating that high-quality, high-coverage test suites still lead to overfitting, and identifying other relationships between test suite properties and patch quality. Finally, prior human evaluations of automatically generated patches have measured acceptability [26] and maintainability [19]. While the human judgment is a criterion not used by the repair tools for patch construction, it is fundamentally different from the correctness criterion we use in our evaluation, as it is often difficult for humans to spot bugs even when told exactly where to look for them [41].

3. THE DATASET

This section describes our dataset of 956 bugs in versions of six small C programs, together with two types of tests and human-written bug fixes. This dataset is available at:
`http://repairbenchmarks.cs.umass.edu/IntroClass/`

3.1 The subject programs

Our dataset is drawn from an introductory C programming class at UC Davis with an enrollment of about 200 students. The use of this anonymized dataset for research was approved by the UC Davis IRB. To prevent identity recovery, students' names in the dataset were securely hashed, and all code comments were removed.

The dataset includes six programming assignments (Figure 1). Each assignment requires students to individually write a program that satisfies a provided set of requirements. The requirements were of relatively high quality: A good deal of effort was spent to make them as clear as possible, given their role in a beginning programming class. Further, the students were taught to first understand the requirements, then design, then code, and finally test their submissions.

Students working on their assignments submit their code by pushing to a personal git repository. The students may submit as many times as they desire without penalty until the deadline. On every submission, a system called GradeBot runs the student program against a set of black-box test cases (described next), comparing the output against an instructor-written reference implementation. The students learn how many tests run and how many pass, but

program	LoC	tests bb	tests wb	buggy versions bb	buggy versions wb	computation
checksum	13	6	10	29	49	checksum of a string
digits	15	6	10	80	140	digits of a number
grade	19	9	9	226	224	grade from score
median	24	7	6	166	153	median of 3 numbers
smallest	20	8	8	153	118	min of 4 numbers
syllables	23	6	10	108	126	count syllables
total	114	42	53	762*	810*	

*956 of the 762 bb and 810 wb buggy versions are unique.

Figure 1: The instructor-written implementations of the six subject programs vary in size from 13 to 24 LOC. The black-box (bb) tests are instructor-written to cover the specification. The white-box (wb) tests are automatically generated for complete coverage of a reference implementation. The programs' revision histories contain 762 versions that pass at least one and fail at least one bb test, and 810 versions that pass at least one and fail at least one wb test, with a total of 956 unique buggy versions.

no other information. The grade is proportional to the number of tests the latest submission (before the deadline) passes. Students do *not* know the test cases used by the GradeBot, so when a submission fails a test, the student has to carefully reconsider the program requirements.

The *G&V* techniques evaluated in this paper rely on a pool of candidate source code elsewhere in the program. We were initially concerned that the programs' small size will impede patch construction. However, as Section 4.1 shows, automated repair was often able to produce patches. Further, we found that increasing the pool of candidate source code lines showed neither an increase in repair rate nor a decrease in overfitting behavior.

3.2 Test suites and measure of patch quality

Each program has two test suites: a black-box test suite and a white-box test suite. The instructor-written *black-box test suite* is based solely on the program specification. The instructor separated the input space into equivalence partitions and selected an input from each partition. The *white-box test suite* achieves edge coverage (also called branch and structural coverage) on the instructor-written reference implementation. We created the white-box test suite using KLEE, a symbolic execution tool that automatically generates tests that achieve high coverage [11]. When KLEE failed to find a covering test suite, we manually added tests to achieve full edge coverage.

The black-box and white-box test suites were developed independently and independently describe desired program behavior. Because students can query how well their submissions do on the black-box tests (without learning the tests themselves), they can use the results of these tests to guide their development. A repair tool can analogously use the black-box tests to guide automated repair.

We use the two test suites to measure functional patch quality. When a human or a tool uses black-box tests to construct a patch, we evaluate how well the patch performs on the held-out white-box test suite. If the patch passes all black-box tests provided as input to the repair tool but fails some white-box tests, then the patch *overfits* to the black-box tests, and fails to generalize to the held-out tests. We can similarly measure overfitting to white-box tests. Several experiments described in Section 4 use this method for measuring patch quality in terms of overfitting and generalizability (the inverse of overfitting).

3.3 Buggy program versions

Because the homework is submitted to a git repository, student submissions to GradeBot provide a detailed history of student efforts to solve each problem. Inevitably, some submissions contain bugs, in that they do not satisfy all of the requirements for the assignment. We can approximate if a submission is buggy by evaluating its performance on the two test suites. Many, though not all, of the final submitted versions are correct. To identify a specific buggy program version, we pick a test suite (e.g., black-box) and find all versions that pass at least one and fail at least one test in that suite. Overall, we identified 762 buggy versions using the black-box suites, and 810 buggy versions using the white-box suites (Figure 1); the union of these sets constitutes 956 unique buggy programs.

For each of the 956 versions, we ran each test and observed the version's behavior on that test. We observed 8,884 failures. The overwhelming majority of errors were caused by incorrect output; this accounted for 8,469 cases. Segmentation faults accounted for 76 test failures; other errors detected by program exit status codes accounted for 254 errors. The remaining 85 errors were due to timeouts, likely caused by infinite loops.

4. EMPIRICAL EVALUATION

We evaluate *G&V* repair via a series of controlled experiments using the dataset from Section 3. Section 4.1 outlines our experimental procedure and reports baseline results for successful patching. Section 4.2 examines overfitting in *G&V* repair and measures how various factors affect overfitting. Section 4.3 compares *G&V* repair to novice developers in terms of overfitting. Finally, Section 4.4 tests previously proposed approaches to combat overfitting.

4.1 Evaluation methodology

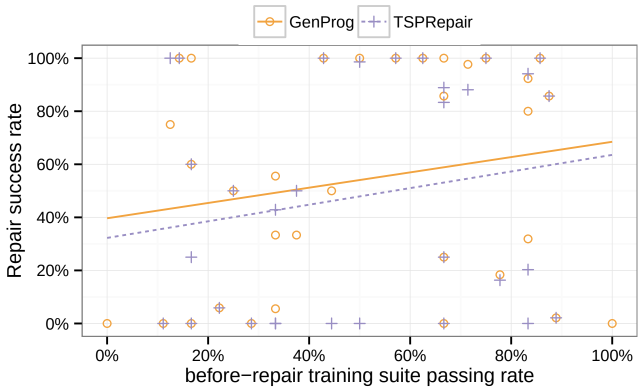
This section outlines the methodology we use to evaluate GenProg and TSPRepair and presents baseline results. We use each tool to attempt to repair each of the 762 program versions that fail at least one black-box test, providing the black-box test suite as the training suite to both tools. For each buggy version, we compute the black-box tests it passes and fails, and then sample randomly those tests to produce 25%, 50%, 75%, and 100% subsets of the training suite of the same pass-fail ratio (rounding up to the nearest test). These test suite subsets represent test suites of varying levels of coverage. We use the term *scenario* to refer to the pair consisting of the buggy program version and a coverage measure. Thus for black-box tests, there are $762 \times 4 = 3,048$ scenarios. We attempt to repair each scenario 20 times, providing a new randomly generated seed each time, for a total of $3,048 \times 20 = 60,960$ attempted repairs. When a tool exits successfully after generating a patch that passes 100% of the training suite, we run the (white-box) held-out evaluation suite over the patch to measure its quality.

Figure 2(a) summarizes the fraction of the time each run, each scenario, and each buggy version was fixed by each of the two tools. While fewer GenProg runs find patches (25.8% vs. 31.2%), GenProg is able to patch more scenarios (46.1% vs. 42.5%) and more distinct buggy program versions (61.9% vs. 57.1%) than TSPRepair.

Figure 2(b) shows the relationship between the number of black-box tests the un-patched buggy program fails and patch success. GenProg is slightly more likely to patch buggy versions that fail fewer tests: A linear regression confirms a slight positive trend (with significance, $p = 0.0106$). The trend detected for TSPRepair is not statistically significant at $\alpha = 0.05$ ($p = 0.0624$, and thus at $\alpha = 0.1$, the result is considered significant). Based on these results, we conclude that GenProg and TSPRepair generate patches sufficiently

tool	runs	scenarios	buggy programs
GenProg	$\frac{15739}{60960} = 25.8\%$	$\frac{1404}{3048} = 46.1\%$	$\frac{472}{762} = 61.9\%$
TSPRepair	$\frac{19023}{60960} = 31.2\%$	$\frac{1294}{3048} = 42.5\%$	$\frac{435}{762} = 57.1\%$

(a)



(b)

Figure 2: (a) GenProg and TSPRepair patch creation rates. (b) GenProg's and TSPRepair's scenario patch creation rates (producing at least one patch that passes all the black-box tests in 20 attempts on different seeds) improve as the number of passing before-repair training suite tests increased. This relationship is significant for GenProg ($p = 0.0106$) but not for TSPRepair.

often to enable further empirical experiments.

All relationships reported in the following sections are evaluated via linear regression, unless otherwise specified. While we give significances where appropriate, none of the detected relationships had large effects measured by R^2, and we do not conclude that any of the relationships are strongly linear.

4.2 Overfitting

> **Research Question 1:** How often do the patches produced by *G&V* techniques overfit to the training suite, and fail to generalize to the held out evaluation suite, and thus ultimately to the program specification?

Having shown that the repair techniques often find patches that cause a program to pass all of the training test suite, we next evaluate the quality of those patches. Specifically, we are interested in learning if *G&V* techniques produce patches that overfit to the training test suite.

We find that the median GenProg patch (which passes 100% of the training suite, by definition) passes only 83.3% of the evaluation suite (mean 83.5%). The median TSPRepair patch passes 67% of the evaluation suite (mean 65%).[1]

We conclude that tool-generated patches often overfit to the training suite used in constructing the patch. For programs that are

[1]For completeness, we also evaluated if AE [57], another publicly available *G&V* tool, overfits. AE produces patches for 33.7% of buggy programs, and the median patch passes 62.5% of the evaluation suite (mean 61.9%). Because AE is deterministic and only produces one patch per buggy program version, our other experiments that rely on using multiple random seeds do not apply.

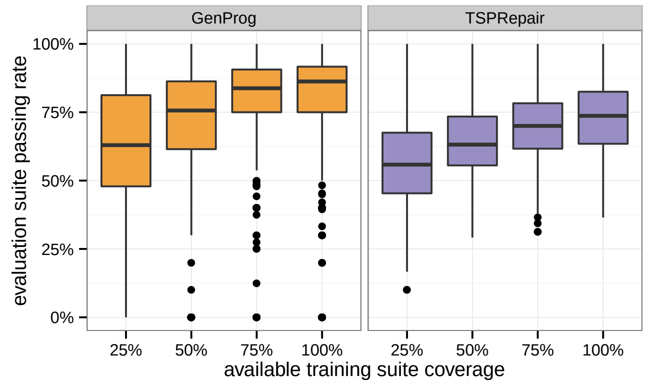


Figure 3: The coverage of the test suite GenProg and TSPRepair use to repair the buggy program strongly correlates ($p < 0.001$) with the portion of the white-box tests the patched program passes.

mostly correct to begin with, both GenProg and TSPRepair are more likely to *decrease* the correctness of the program under repair than to increase it.

> **Research Question 2:** How does training suite coverage affect patch overfitting?

In practice, test suites are typically incomplete. To measure how *G&V* techniques perform when given incomplete test suites, we use subsets of the black-box test suites as the training suites, and measure the relationship between the coverage of the training suite and the patch's overfitting.

For each buggy program, we use the test suite sampling procedure from Section 4.1 to produce 25%-, 50%-, 75%-, and 100%-sized test suites that keep consistent the pass-fail ratio of every buggy version, but vary the test suite coverage. As before, for each tool, we repeat this process 20 times, each time resampling the test suites and using a different random seed.

Figure 3 shows the relationship between training suite coverage and overfitting (the fraction of the held-out white-box tests the patched program passes). For both GenProg and TSPRepair, higher-coverage training suites improve the quality (reduce the overfitting) of a patch: the patch passes more white-box tests, on average. A linear regression confirms both positive trends (with significance, $p < 0.001$).

We conclude that GenProg and TSPRepair benefit from high-coverage test suites in repairing bugs. Using low-coverage test suites, which are unfortunately common in practice, poses a risk of automated patches that overfit to that test suite.

> **Research Question 3:** How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit?

Section 4.1 showed that the number of training tests the buggy version fails is related to a technique's ability to produce a patch. Now, we explore if it is also related to overfitting.

Figure 4 relates the quality of the generated patch as measured by its performance on the held-out white-box tests to the number of training black-box tests the original program passes. The top of Figure 4 shows that programs that pass more training tests before

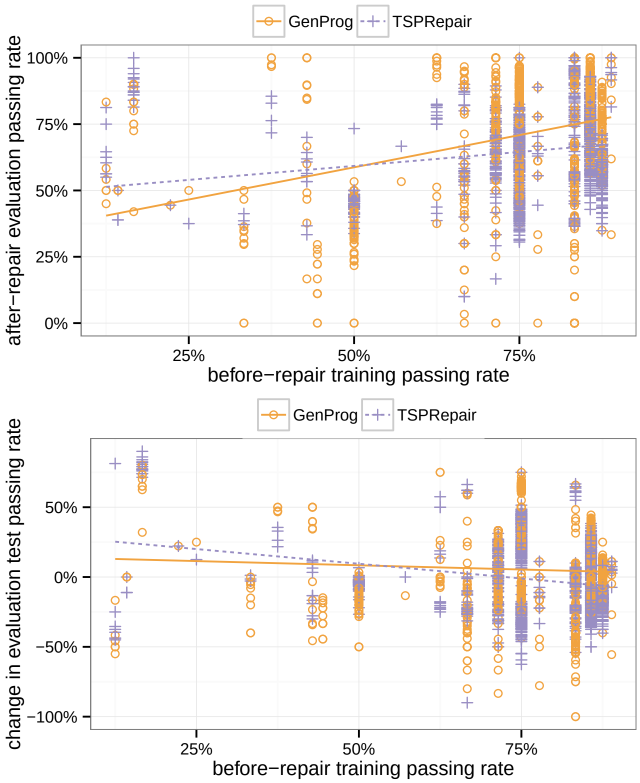


Figure 4: Top: The fraction of evaluation tests the patched program passes is significantly positively correlated with the fraction of training tests the un-patched version passes (p < 0.001 for both tools). Bottom: However, for un-patched programs that pass more of the training tests to start with, both tools are more likely to break functionality than fix it; the correlation between before-repair training suite pass rate and evaluation tests fixed is significantly negative (p < 0.001 for both tools).

repair are more likely to pass the evaluation tests post-repair. Linear regression confirms the positive trend for both tools with significance, $p < 0.001$. However, the bottom of Figure 4 shows that both GenProg and TSPRepair are also more likely to break the held-out test cases than fix them when repairing programs that initially pass most of the black-box tests. Again, a linear regression confirms the negative trend for both tools with significance, $p < 0.001$.

We conclude that *G&V* repair presents a danger when fixing high-quality programs that pass most of their test suites. The patches are likely to overfit to the tests, breaking other, previously correct functionality. For low-quality programs that fail many tests, GenProg and TSPRepair repair more functionality than they break, on average.

> **Research Question 4:** How does the training test suite's provenance (automatically generated vs. human-written) influence the patches' overfitting?

We have shown that using low-coverage test suites to fix bugs can lead to low-quality patches. This suggests that automatic test generation might be used to improve test suite coverage prior to a repair attempt. Here, we evaluate if automatically generated tests (generated with KLEE [11] as described in Section 3.2) are as effective for use by *G&V* repair as human-written tests. We refer to the method by which the tests are created as *test provenance*.

Figures 5(a) and 5(c) summarize the relationship between test suite provenance and GenProg patch overfitting. When GenProg repaired buggy programs using (all of) the black-box tests as the training suite (Figure 5(a)), its patches did relatively well on the white-box evaluation tests. However, the same was not true when GenProg used (all of) the white-box test suite as the training suite, with the black-box tests as the held-out suite (Figure 5(c)). In the latter case, GenProg overfit significantly to the white-box tests. Figure 5(e) directly compares the two provenance methods. A two-sample test supports our conclusion that the black-box patches pass more of the white-box tests than the white-box patches do the black-box tests (with significance, $p < 0.001$). Cliff's Delta test reports a large magnitude effect (magnitude > 0.5).

Similarly, Figures 5(b) and 5(d) summarize the effect of test suite provenance on TSPRepair patch quality, and Figure 5(f) directly compares the two provenance methods. The effect is nearly identical to GenProg, although TSPRepair has slightly worse performance, even with black-box tests. The two-sample test similarly supports this conclusion ($p < 0.001$), and a Cliff's Delta test reports a similar large magnitude effect.

We conclude that test suite provenance plays an important role in GenProg-generated patch quality. Not all full-coverage test suites are created equal, and some are better suited for automated repair.

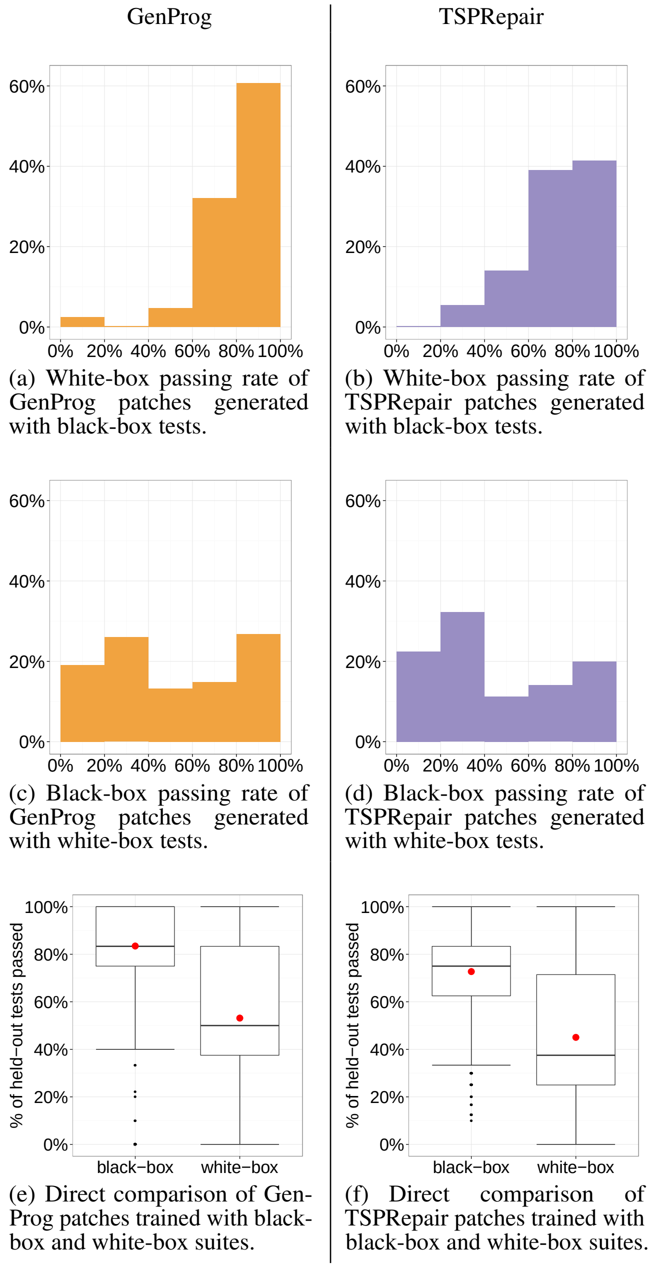
4.3 Do tools outperform novice developers?

One of the advantages of our dataset is that every program has a human fix associated with it, corresponding to the student's final submission. The students who produced the programs in our dataset are faced with a challenge similar that presented to our repair tools. They write and submit code, gain information about how many tests their code passes and fails, make changes, and resubmit. Those who have taught introductory programming courses know that students follow a number of search strategies while constructing repairs, ranging from structured reasoning to random search. This section compares the patches produced by the automated repair tools to the results of novice developers' repair attempts. As before, repair tools (and now, humans) have the black-box test suite available during repair to serve as a training suite, and the white-box tests are held out and can be used to evaluate the quality of the repair.

> **Research Question 5:** Do tool-generated patches overfit more than novice-developer-written patches?

Figure 6(a) shows that student solutions do, in fact, overfit to the provided test suites and often fail to generalize to held-out tests. Figure 6 compares the quality of GenProg, novice developer, and TSPRepair patches. The mean GenProg-generated patch trained on the same (black-box) test suite is of higher quality than those created by the students (although the median student patch is of higher quality than the median GenProg-generated one). The Wilcoxon signed rank test supports a significant ($p < 0.001$) difference between the GenProg-generated and human-written patches (Figure 6(b)). However, the improvement is only slight, and the Cliff's Delta test indicates that the effect size is negligible. While the GenProg-generated patches are only slightly better, they also demonstrate significantly less quality variability than student-written patches. This is also evident in Figure 6(b).

TSPRepair patches have both a lower mean and median passing rate for held-out tests than the student-written and GenProg-generated patches. While there is a visual difference in Figure 6(b) between student-written and TSPRepair-generated patches, the Wilcoxon signed rank test reports no significant difference between the



(a) White-box passing rate of GenProg patches generated with black-box tests.

(b) White-box passing rate of TSPRepair patches generated with black-box tests.

(c) Black-box passing rate of GenProg patches generated with white-box tests.

(d) Black-box passing rate of TSPRepair patches generated with white-box tests.

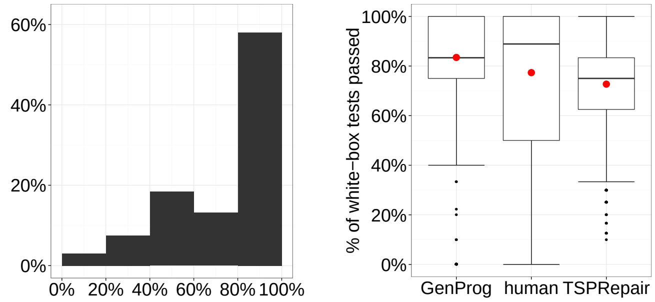
(e) Direct comparison of GenProg patches trained with black-box and white-box suites.

(f) Direct comparison of TSPRepair patches trained with black-box and white-box suites.

Figure 5: (a) and (b): When black-box tests guided repair search, the resulting patches did well on the evaluation white-box tests. (c) and (d): However, the same was not true when using white-box tests to guide the search for patches. (e) and (f): The direct comparisons show that patches generated using the black-box suite generalize to evaluation tests much better than patches generated using the white-box suite. (The line shows the median, and the dot the mean.) For both tools, Wilcoxon signed-rank tests detected a significant difference, p $<$ 0.001 with a large Cliff's Delta in both cases.

samples. As with GenProg, TSPRepair-generated patches demonstrate significantly less variability in quality than student-written patches.

Comparing automatically generated patches to novice-developer-written patches might seem unfair, since repair tools can only access



(a) White-box passing rate of novice-developer-written patches using black-box tests.

(b) Direct GenProg, novice developer, and TSPRepair comparison.

Figure 6: (a): Novice-developer-written patches also overfit to the black-box tests used during development. (b) The median (shown as the line) human-written patch overfits slightly less than those generated by GenProg and TSPRepair, but the mean (shown as the dot) GenProg-generated patches overfit slightly less than the others, in part because the student-written patches show higher variance than both automatic techniques. A Wilcoxon signed rank test (p $<$ 0.001) supports this conclusion about GenProg; the same test for TSPRepair does not reject the null hypothesis with significance.

tests that represent a partial specification, while humans can reason abstractly about the program specification. However, while humans can reason about program faults abstractly above the level of a repair tool, they are also subject to a large array of cognitive biases [1, 34] that can hamper their debugging effort. Repair tools have no such biases, and will mechanically explore the solution space as guided by their fitness function, without becoming irrationally fixated on particular solutions.

4.4 Mitigating overfitting

Research Question 6: Does minimizing the GenProg-generated patches affect the specificity of the patches?

GenProg uses patch minimization, via delta debugging [59], to reduce code bloat. TSPRepair does not perform minimization, because the produced patch is only ever a single edit. Intuitively, a small change to a program is less likely to encode special behavior that handles just the training tests in a separate way [58]. Thus far, all results we have described for GenProg have used GenProg's built-in patch minimization procedure. We now investigate if disabling this feature increases the overfitting.

We compared unminimized patches produced by GenProg to their minimized versions in terms of the number of black-box and white-box tests the patched versions passed. In all experiments, regardless of the tests used, paired Wilcoxon tests show that the test-passing rates of the minimized and unminimized patches were drawn from the same distribution, and fail to reject the null hypothesis ($p > 0.1$ in all cases, after Benjamini-Hochberg correction for false discovery rates). This indicates that minimization does not reduce the degree to which GenProg overfits.

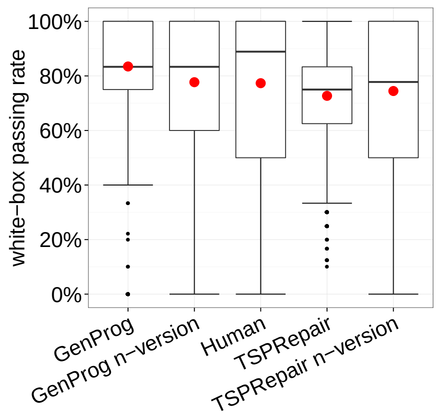


Figure 7: Tool-generated patches and n-version programs made up of those patches perform worse than humans-written patches, on average. N-version GenProg programs underperform even the individual GenProg patches, and n-version TSPRepair programs perform negligibly worse than individual TSPRepair patches while not statistically differing from human-written patches.

> **Research Question 7:** Can overfitting be *averaged out* by exploiting randomness in the repair process? Do different random seeds overfit in different ways?

Some repair tools, including GenProg and TSPRepair, can generate multiple patches for the same defect (such as when run on multiple different random seeds). This affords a unique opportunity: Even if patches do overfit to their test suites, it is possible that a group of patches better represents the desired program behavior than an individual patch. Specifically, even if each patch overfits on some subset of desired behavior, if each patch in a group encodes most of that behavior, a group vote on the behavior may outperform each individual patch. N-version patches may therefore provide an avenue to mitigate overfitting. Human-written code typically lacks sufficient diversity [27] to enable true n-version programming [14], but randomized *G&V* repair may not.

To create the n-version program $\mathcal{P}_n$, we: For each buggy version-test suite subset pair $\mathcal{P}_b$, run GenProg on $\mathcal{P}_b$ 20 times. If fewer than three of the runs result in a patch, we exclude this pair from this experiment. We call these ($n \geq 3$) patched versions $\mathcal{P}_p^1 \dots \mathcal{P}_p^n$. Next, we create a new program, $\mathcal{P}_n$, that on input i, runs each of $\mathcal{P}_p^1 \dots \mathcal{P}_p^n$ on i, and returns the output most frequently returned by output by those program. If two or more return values tie, $\mathcal{P}_n$ returns one of those values at random.

Figure 7 shows that n-version patches constructed from GenProg's output do not perform statistically significantly better than either individual GenProg-generated patches or novice-developer-written patches. While n-version TSPRepair patches are statistically significantly better than individual TSPRepair patches ($p < 0.001$), the Cliff's Delta is negligible, and n-version TSPRepair patches do not significantly outperform those written by novice developers. The only other case in which n-version programs outperformed individual patches was when GenProg constructed patches using white-box suites for training (not shown in Figure 7). Recall that training on white-box suites produced poor-quality patches (Research Question 4). In this case, the GenProg n-version patches significantly outperform individual patches ($p < 0.001$), but the Cliff's effect size is small.

We conclude that when tools can produce quality patches (using high-quality test suites), there is insufficient diversity in the patches to further improve quality. However, when repair tools produce poor quality patches, diversity sometimes provides a modest benefit. N-version programming may indeed provide an avenue to mitigate the worst cases of overfitting.

5. CASE STUDY

Section 4.2 showed that test suite provenance has the largest effect on the quality of automatically generated patches. This section describes a case study of a buggy student program and two patches that GenProg produced for it using the white-box test suite to highlight the ways that some test suites can lead to increased overfitting.

The `median` homework assignment asks students to produce a C function that takes as input three integers and outputs their median. Figure 8 shows the black- and white-box test suites for the `median` program.

One of the student's buggy (non-final) submissions to the homework was:

```
1  int med(int n1, int n2, int n3) {
2    if ((n1==n2) || (n1==n3) ||
3          (n2<n1 && n1<n3) || (n3<n1 && n1<n2))
4      return n1;
5    if ((n2==n3) || (n1<n2 && n2<n3) ||
6               (n3<n2 && n2<n1))
7      return n2;
8    if (n1<n3 && n3<n2)
9      return n3;
10 }
```

This submission is close to correct. Despite its incorrect logic (e.g., the equality checks on lines 2 and 5), it passes five of the six white-box and six of the seven the black-box tests. It does not return an answer for the fifth black-box and for the second white-box tests, for which `n3` is the median and `n1 > n2`.

Given this program and the white-box suite, GenProg generated several patches of varying quality. One such low-quality, GenProg-patched program is:

```
1  int med(int n1, int n2, int n3) {
2    if ((n1==n2) || (n1==n3) || ((n3<n1) && (n1<n2)))
3      return n1;
4    if (n2<n1)
5      return n3;
6    if ((n2==n3) || ((n1<n2) && (n2<n3)) ||
7             ((n3<n2) && (n2<n1)))
8      return n2;
9    if ((n1 < n3) && (n3 < n2))
10     return n3;
11 }
```

One of the conditions in the check on line 2 has been removed, and this program returns `n1` as the median if it is coincidentally equal to either `n2` or `n3`, or if it is actually the median and `n3 < n2`. If `n1` is not the median, but `n2 < n1` (the check moved to line 5), this code will (possibly, but not necessarily incorrectly) return `n3`. The rest of the logic is unaffected.

This patch addresses the original problem in the student's code, at least with respect to the white-box suite. This code is correct when `n1` is the median and `n3 < n2`, `n2` is the median and `n2 > n1`, or `n3` is the median and `n2` $\leq$ `n1`. Although this code passes all of the white-box tests (improving on the original student submission), it passes fewer black-box tests than the original, failing tests 3 and 6 in Figure 8.

black-box tests	white-box tests
`med(2, 6, 8) = 6`	`med(0, 0, 0) = 0`
`med(2, 8, 6) = 6`	`med(2, 0, 1) = 1`
`med(6, 2, 8) = 6`	`med(0, 0, 1) = 0`
`med(6, 8, 2) = 6`	`med(0, 1, 0) = 0`
`med(8, 2, 6) = 6`	`med(0, 2, 1) = 1`
`med(8, 6, 2) = 6`	`med(0, 2, 3) = 2`
`med(9, 9, 9) = 9`	

Figure 8: White- and black-box suites for `median`.

This patch is an excellent example of overfitting the fitness function, and highlights weaknesses in the white-box test suite: Many of the inputs have repeated elements. As a result, the student's otherwise logically incorrect equality checks on lines 2 and 5 of the original submission mask the larger problems in the low-quality patch.

Running GenProg with the same white-box test suite but a different random seeds can lead to different patches for the same bug. For example, for this buggy program, GenProg also produced the following patched program:

```
1   int med(int n1, int n2, int n3) {
2     if ((n1==n2) || (n1==n3) || ((n2<n1) && (n1<n3)) ||
          ((n3<n1) && (n1<n2)))
3       return n1;
4     if ((n2==n3) || ((n1<n2) && (n2<n3)) ||
5         ((n3<n2) && (n2<n1)))
6       return n2;
7     if ((n1 < n3) && (n3 < n2))
8       return n3;
9     else
10      return n3;
11  }
```

The incorrect equality checks on lines 2 and 4 remain. This patch inserted `return n3` into the `else` block of the last set of conditions that seek to determine if `n3` is the median. Ignoring the equality checks, this is actually a reasonable solution, because by that point, the only remaining option *should* be that `n3` is the median.

For this buggy program, the student rewrote the logic considerably, eliminating the equality checks on lines 2 and 4 and properly handling the last set of conditionals:

```
1   int med(int n1, int n2, int n3) {
2     if ((n2<=n1 && n1<=n3) || (n3<=n1 && n1<=n2))
3       return n1;
4     if ((n1<n2 && n2<=n3) || (n3<=n2 && n2<n1))
5       return n2;
6     if ((n1<n3 && n3<n2) || (n2<n3 && n3<n1))
7       return n3;
8   }
```

In this example, GenProg solutions overfit to the test suite, while the student-written patch is more general. This example highlights weaknesses in the white-box test suite, which fails to encode key behavior. This raises interesting questions about the potential of automatic test case generation to augment the input given to *G&V* repair techniques; more work is required to improve the quality of the output of such techniques before the two approaches can be usefully integrated.

6. THREATS TO VALIDITY

Our experiments may not generalize. We only experiment with GenProg and TSPRepair, two of several *G&V* repair techniques, and our results may not extend to other automatic program repair mechanisms. However, recent work has started to unify the theory underlying *G&V* repair [57], suggesting that results from two different techniques may extend to others. Our subjects are small student-written programs, with fairly small test suites. Therefore, our results may not generalize to large, real-world programs. However, this is a necessary tradeoff, as the goals of our study require exhaustively testable programs with multiple exhaustive test suites. Understanding repair techniques at the scale of our experiments increases understanding of the repair techniques in general. Additionally, while our subjects' size allows for a very large dataset for conducting controlled trials, it may also affect the ability to find diverse patches. We ran 20 seeds per repair effort, a relatively small number by the standards of metaheuristic search algorithms. More attempts may have revealed more solutions. Finally, we used the recommended GenProg parameter set defined in previous work [30]; a full parameter sweep is outside the scope of this investigation.

We release our dataset, including all the buggy versions, student-written solutions, and test suites. This makes our experiments repeatable. However, parts of the creation of the dataset were manual. While the white-box suites were generated automatically to the extent possible, and black-box suites were generated by a rigorous manual analysis of the requirements, at least the latter is subject to human interpretation. Thus, a replication of our experiments on different programs or with different test suites on our programs may be affected by human subjectivity and may produce different results.

GenProg, TSPRepair, and many other related repair techniques rely on randomized algorithms. Evaluating systems that involve randomized algorithms is particularly difficult and requires paying special attention to the sample sizes, statistical tests, cross-validation, and uses of bootstrapping. Our work is consistent with the guidelines for evaluating randomized algorithms [4] to enhance the credibility of our findings. Specifically, we used a large sample of 956 buggy student programs, controlled for a variety of potential influencers in our experiments, and used fixed-effects regression models and two sample tests along with false-discovery rate correction to lend statistical support to our findings.

7. RELATED WORK

Our work evaluates automated repair so that it can be improved. Empirical studies of fixes of real bugs in open-source projects can also improve repair by helping designers select change operators and search strategies [25, 60]. Understanding how automated repair handles particular classes of errors, such as security vulnerabilities [31, 43] can guide tool design. For this reason, some automated repair techniques focus on a particular defect class, such as buffer overruns [51, 53], unsafe integer use in C programs [16], single-variable atomicity violations [24], deadlock and livelock defects [32], concurrency errors [33], and data input errors [3]. Other techniques tackle generic bugs. For example, the ARMOR tool replaces buggy library calls with different calls that achieve the same behavior [12], and *relifix* uses a set of templates mined from regression fixes to automatically patch generic regression bugs. Our evaluation has focused on tools that fix generic bugs, but our methodology can be applied to focused repair as well.

User-provided code contracts, or other forms of invariants, can help to *synthesize* correct-by-construction patches, e.g., via AutoFix-E [42, 56] (for Eiffel code) and SemFix [39] (for C). DirectFix [35] aims to synthesize minimal patches to be less prone to overfitting, but only works for programs with a subset of the language features, and has only been tested on small programs. These techniques have the benefit of correctness proofs, but require contracts, so they are unsuitable for legacy systems. Synthesis techniques can also construct new features from examples [15, 21], rather than address existing bugs. Our work has focused on *G&V* approaches, and investigating overfitting and patch quality in synthesis-based techniques is a complementary and worthwhile pursuit.

The techniques evaluated in this paper, GenProg and TSPRepair, are representative of *G&V* approaches. Our work does not create a new bug-fixing technique, but rather evaluates existing techniques in a new way to expose previously hidden limitations to *G&V* program repair. Our findings may extend to other search-based or test suite-guided repair techniques (e.g., [5, 17, 26, 35, 39, 40, 43, 57]). Section 2.3 has already discussed previous evaluations of *G&V* techniques. Monperrus [38] has recently discussed the challenges of experimentally comparing program repair techniques. For example, the selection of test subjects (defects) can introduce evaluation bias [9, 44]. Our evaluation focuses precisely on the limits and potential of repair techniques on a large dataset of defects, and controls for a variety of potential influencers, addressing some of Monperrus' concerns [38].

Genetic programming tends to produce extraneous code that does not contribute to the fitness of the solution [22, 52] and may lead to overfitting. GenProg attempts to mitigate this through solution minimization. Overfitting is also a well-studied problem in machine learning [37]. Our experiments suggest that minimization and overfitting are unrelated, which is consistent with prior results in machine learning [49]. To the best of our knowledge, ours is the first consideration of this relationship in the program repair domain.

G&V approaches fall in the space of search-based software engineering [23], which adapts search methods, such as genetic programming, to software engineering tasks. Search-based software engineering has been used for developing test suites [36, 55], finding safety violations [2], refactoring [50], and project management and effort estimation [7]. Good fitness functions are critical to search-based software engineering. Our findings indicate that using test cases alone as the fitness function leads to patches that may not generalize to the program requirements, and more sophisticated fitness functions may be required for search-based program repair.

N-version programming [14] combines multiple different programs trying to solve the same problem in the interest of achieving resiliency and correctness through redundancy. N-version programming works poorly with human-written systems because the errors humans make do not appear to be independent [27]. Our evaluations have shown that n-versions of automatically generated patches has a minor positive effect but failed to fully generalize to the desired behavior.

8. CONCLUSIONS AND IMPLICATIONS

G&V automated repair shows promise for reducing the manual bug-fixing burden and improving software quality. However, if these techniques are to gain practical traction, we must augment feasibility demonstrations with qualitative evaluations that address the quality and applicability. In this paper, we systematically evaluated the factors affecting the output quality of GenProg and TSPRepair, two representative *G&V* techniques, through a controlled evaluation on a large set of programs written by novice developers with naturally occurring bugs and human-written patches. Based on our findings, the open research challenges include:

Repair techniques must go beyond testing on the training data to characterize functional correctness. GenProg and TSPRepair produced patches for more than half of the bugs in our dataset (61.9% and 57.1%, respectively). The ability to produce a patch was correlated with input program quality, as measured by the test suites. However, those patches tended to overfit to the test suite used to generate the patch. Interestingly, the novice programmers (students) also overfit to the provided test cases. When using requirements-based (black-box) tests, GenProg overfit less than the students did. These results highlight both the significant promise of automatic repair and the fact that more work is needed to improve repair output quality.

We propose that future evaluations of *G&V* repair tools withhold some portion of tests from the repair tool, at least some of which share code-under-test with the tests exposing the buggy behavior. This is similar to the machine learning evaluational technique of *cross-validation*, and provides a higher level of confidence that a repair technique is able to repair isolated defects without introducing regressions.

Automatic repair should be used in appropriate contexts. Both test suite coverage and input program quality appear related to the quality of the automatically generated patches. Higher-coverage test suites were more likely to lead to more general patches, while patches produced for higher-quality programs were more likely to break existing functionality. This suggests that automatic repair techniques might be best applied early in the development lifecycle, though unfortunately, this is the time when the program quality itself is likely low (reducing the likelihood of repair success), and the test suite is least likely to be comprehensive. Different repair techniques are likely to be useful at different times, and more study is needed to explore this space.

The quality of repair test suites should be measured and improved appropriately. The provenance of the test suites — automatically-generated or human-written — had a striking relationship with the resulting patch quality. Automatic test-input generation techniques should fit naturally into a toolchain for automatic repair, particularly when user-provided test cases fail to fully cover the program functionality, or when critical functionality should be independently tested post-repair, to ensure that overfitting has not occurred. Our results suggest that more work is needed to fully understand and characterize test suite quality beyond coverage metrics alone.

Patch diversity might improve repair quality. Low-quality patches, especially those generated using automatically generated tests, demonstrated sufficient functional diversity to improve on the patched programs via plurality voting. Plurality voting may thus mitigate the risks of low-quality test suites, in the appropriate settings.

While *G&V* techniques have not yet become a silver bullet of program repair, in some cases and settings, they already outperform beginner developers. Our results suggest that if several shortcomings are addressed, there is significant promise that automated repair techniques can be impactful and helpful parts of the software development process.

Acknowledgments

Prem Devanbu and Ming Xiao were instrumental in the creation of an earlier version of the student programs dataset and early GenProg experiments [10].

9. REFERENCES

[1] R. E. Adamson. Functional fixedness as related to problem solving: A repetition of three experiments. *Journal of Experimental Psychology*, 44(4):288–291, 1952.

[2] E. Alba and F. Chicano. Finding safety errors with ACO. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, pages 1066–1073, London, England, UK, 2007.

[3] M. Alkhalaf, A. Aydin, , and T. Bultan. Semantic differential repair for input validation and sanitization. In *International Symposium on Software Testing and Analysis*, pages 225–236, 2014.

[4] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1–10, Honolulu, HI, USA, 2011.

[5] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Congress on Evolutionary Computation*, pages 162–168, 2008.

[6] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 306–317, Hong Kong, China, November 2014.

[7] A. Barreto, M. Barros, and C. Werner. Staffing a software project: a constraint satisfaction approach. *Computers and Operations Research*, 35(10):3073–3089, 2008.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[9] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 121–130, Amsterdam, The Netherlands, August 2009.

[10] Y. Brun, E. Barr, M. Xiao, C. Le Goues, and P. Devanbu. Evolution vs. intelligent design in program patching. Technical Report `https://escholarship.org/uc/item/3z8926ks`, UC Davis: College of Engineering, 2013.

[11] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, USA, 2008.

[12] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 782–791, San Francisco, CA, USA, 2013.

[13] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for web applications. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE10)*, pages 237–246, Santa Fe, New Mexico, USA, 2010.

[14] L. Chen and A. Avižienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, 1978.

[15] R. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes. Program boosting: Program synthesis via crowd-sourcing. In *Symposium on Principles of Programming Languages (POPL)*, pages 677–688, Mumbai, India, January 2015.

[16] Z. Coker and M. Hafiz. Program transformations to fix C integers. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 792–801, San Francisco, CA, USA, 2013.

[17] V. Debroy and W. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation*, pages 65–74, Paris, France, 2010.

[18] H. Estler, C. A. Furia, M. Nordio, M. Piccioni, B. Meyer, et al. Contracts in practice. *arXiv preprint arXiv:1211.4775*, 2012.

[19] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

[20] M. Gabel and Z. Su. Testing mined specifications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, Cary, NC, USA, 2012.

[21] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Symposium on Principles of Programming Languages (POPL)*, pages 317–330, Austin, TX, USA, 2011.

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

[23] M. Harman. The current state and future of search based software engineering. In *International Conference on Software Engineering*, pages 342–357, 2007.

[24] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 389–400, San Jose, CA, USA, 2011.

[25] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, San Jose, CA, USA, July 2014.

[26] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*, pages 802–811, San Francisco, CA, USA, 2013.

[27] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

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

[29] 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 *Proceedings of the 34th International Conference on Software Engineering*, pages 3–13, Zurich, Switzerland, 2012.

[30] C. Le Goues, S. Forrest, and W. Weimer. Representations and operators for improving evolutionary software repair. In *Genetic and Evoluationary Computation Conference*, pages 959–966, 2012.

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

[32] Y. Lin and S. S. Kulkarni. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In *International Symposium on Software Testing and Analysis*, pages 237–247, 2014.

[33] P. Liu, O. Tripp, and C. Zhang. Grail: context-aware fixing of concurrency bugs. In *International Symposium on Foundations of Software Engineering*, pages 318–329. ACM, 2014.

[34] A. S. Luchins. Mechanization in problem solving: The effect of Einstellung. *Psychological Monographs*, 54(6):i–95, 1942.

[35] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the ACM/IEEE 37st International Conference on Software Engineering (ICSE15)*, Florence, Italy, 2015.

[36] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[37] T. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[38] M. Monperrus. A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *International Conference on Software Engineering*, pages 234–242, 2014.

[39] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781, San Francisco, CA, USA, 2013.

[40] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *Trans. Evol. Comp*, 15(2):166–182, Apr. 2011.

[41] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 199–209, Toronto, ON, Canada, 2011.

[42] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering (TSE)*, 40(5):427–449, 2014.

[43] 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, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 87–102, Big Sky, MT, USA, October 12–14, 2009.

[44] D. Posnett, V. Filkov, and P. Devanbu. Ecological inference in empirical software engineering. In *International Conference on Automated Software Engineering (ASE)*, pages 362–371, Lawrence, KS, USA, November 2011.

[45] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *International Conference on Software Maintenance (ICSM)*, pages 180–189, September 2013.

[46] 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 (ICSE)*, pages 254–265, 2014.

[47] Z. Qi, F. Long, S. Achour, , and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. Technical Report MIT-CSAIL-TR-2015-003, MIT Computer Science and Artificial Intelligence Laboratory, 2015.

[48] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. Technical Report NIST Planning Report 02-3, May 2002.

[49] J. Rissanen. Modelling by the shortest data description. *Automatica*, 14:465–471, 1978.

[50] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1909–1916, Seattle, WA, USA, 2006.

[51] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, Nov. 2005.

[52] S. Silva and E. Costa. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179, June 2009.

[53] A. Smirnov and T. cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed Systems Security*, 2005.

[54] S. H. Tan and A. Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the ACM/IEEE 37st International Conference on Software Engineering (ICSE15)*, Florence, Italy, 2015.

[55] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–12, Portland, ME, USA, 2006.

[56] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 61–72, Trento, Italy, 2010.

[57] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th International Conference on Automated Software Engineering*, Palo Alto, CA, USA, 2013.

[58] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the ACM/IEEE 31st International Conference on Software Engineering (ICSE09)*, pages 364–374, Vancouver, BC, Canada, 2009.

[59] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[60] H. Zhong and Z. Su. An empirical study on real bug fixes. In *International Conference on Software Engineering (ICSE)*, Florence, Italy, 2015.