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 search-based automated program repair techniques has shown promise for reducing the significant manual effort required for debugging. 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.* 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 techniques [3, 11, 15, 16, 23, 25, 28, 31, 32, 34, 38, 41, 42, 45, 52, 54, 55] hold 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 [28], while PAR showed comparable results on 119 Java bugs [25]. While some techniques validate patch correctness with respect to user-provided or inferred contracts [23, 38, 41, 54], a larger proportion use test cases. The most common prior evaluations of automatic repair have provided evidence of techniques' feasibility with respect to this test-case-based definition of patch correctness (e.g., [15, 30, 38, 41, 54]).

However, in practice, a test suite is never exhaustive [46], and repair techniques must avoid breaking undertested functionality. However, when repair technique evaluations use the same test cases or workloads to both construct the patch and to validate its correctness, they fail to measure whether the repair technique breaks functionality. In our review of the literature, prior evaluations of automated repair techniques that relied on test cases or workloads all suffered from this deficit, failing to evaluate patch quality independently of the test cases used to construct the patch. Research performed concurrently with ours, e.g., [45, 58] has begun to consider independent measures of quality, though less extensively than we do here. And while some evaluations have used humans to measure repair acceptability [25] and maintainability [18], unlike our work, they did not directly evaluate patch correctness.

We term 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 [23, 41, 54]**[[claire's remembering the other citation to add]]** 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 an accurate, large-scale, controlled investigation of GenProg [30] and TSPRepair [43, 44], both test-case-guided, search-based automatic program repair tools with freely available implementations that scale well to large programs. The evaluation identifies when 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 the patch. 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, specifically checking that it does not break existing functionality. Patches that are overly specific to the training tests and fail to generalize to the held-out tests *overfit* the training tests. Techniques that produce overfitting patches tend to fix certain program behavior while breaking other behavior.

The key 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 test suite used for patch construction on patch quality. This is only possible on small, exhaustively testable programs. 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 be repaired to support a large-scale evaluation that enables 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. We release this dataset to foster better evaluation of future automated repair tools: `http://repairbenchmarks.cs.umass.edu/IntroClass/`. 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 [10].

Using larger, real-world programs written by professional developers is desirable for evaluating scalability and applicability to complex program behavior, but these properties have been evaluated previously (e.g., [28, 42]) and are not our focus. Using small programs is a threat to the generalizability of our results, but this is a trade-off we make because evaluating the properties of repair we are interested in is only possible on programs with exhaustive test suites. Understanding repair techniques at this scale is useful to better our understanding the repair techniques in general.**[[I find this paragraph a bit defensive, and repetitive, possibly move to threats? Would get us to the point a little faster.']]**

To the best of our knowledge, this is the first systematic effort to evaluate the correctness of automated program 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 beginning 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. Since the repair techniques evaluated in this paper rely on a pool of candidate source code lines elsewhere in the program, we were initially concerned that the programs' small size will impede repair construction. However, we found that both techniques were often able to produce patches, and that increasing the pool of candidate source code lines showed neither an increase in repair construction, nor a decrease in overfitting behavior. Our study increases the understanding of how, why, and under what circumstances search-based repair succeeds and fails, which would be more difficult on large, complex programs, where direct, large-scale experimental comparison is dramatically harder. We find that:

- As the number of training test failures increases, the probability of GenProg and TSPRepair producing a patch decreases.
- Both tools overfit to the training test suite used to guide patch construction, often breaking 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 when repairing programs that have fewer defects. In these cases, the "patch" is *worse* than the un-patched program.
- Human-generated black-box, requirement-based test suites are significantly better for use in automatic repair than branch-coverage-maximizing white-box test suites generated using KLEE.
- Novice-developer-written patches also overfit to the test suites, more so than tool-generated patches guided by high-quality black-box test suites, but less so than those guided by the white-box test suites.
- Since GenProg and TSPRepair are nondeterministic, they can generate multiple patches for the same bug. When white-box tests are used as the training suite, combinations of GenProg patches perform statistically significantly better than individual GenProg patches, but in all other cases there is no significant improvement in using a combination of patches. Combinations of patches never outperform human-written patches.

Additionally, we contribute an evaluation methodology and a novel dataset that enables large-scale, controlled evaluations of the factors contributing to the quality of automatic repair.

The rest of this paper is structured as follows. Section 2 discusses automated repair. Section 3 describes our 956-bug dataset. Section 4 conducts 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., [23, 41, 54] **[[claire is adding the other citation]]**). (2) *Generate-and-validate* (*G&V*) techniques create candidate patches (often via search-based software engineering [22]) and then validate them, typically through testing (e.g., [3, 11, 12, 15, 16, 25, 32, 34, 38, 42, 45, 52, 55, 56]. This paper focuses on *G&V* techniques. Section 2.1 defines this class, Section 2.2 explains the reasons for our focus, 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 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, GenProg [28,56], TSPRepair [43], and AE [55] are the only publicly available repair tools that both repair programs written in C and target general-purpose bugs (as opposed to focusing on only one domain of bugs, such as concurrency bugs or integer overflow bugs). In this paper, we use GenProg and TSPRepair as exemplars of *G&V* program repair. (AE is deterministic, which makes it different from GenProg and TSPRepair, so our experimental methodology does not apply to AE, but we do find that AE similarly overfits the tests used for patch construction, as Section 4.2 explains.)

GenProg [28, 56] uses a genetic programming heuristic [27]. Given a buggy program and a set of tests, GenProg generates a population of random patches and uses the number of tests each patch passes as its fitness score to inform a weighted random selection of a subset of the population to propagate into the next iteration of the algorithm. These patches are recombined and mutated to form new candidates using genetic programming mechanisms, until either a patch passes 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 [43] uses random search in place of GenProg's genetic programming approach to traverse the infinitely large search space of candidate solutions. Further, 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 is more efficient than GenProg in terms of time and test case evaluations [43]. The same approach is also called RSRepair [44], and we refer to the original algorithm name in this paper. **[[we never said anything about single edit patches!]]**

There are three key hurdles that *G&V* must overcome to find patches [55]. 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 prioritize changing program constructs that are executed exclusively by failing tests over those that are also executed by passing tests. 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, 19] and logic implemented with a bug in one place is likely to be implemented correctly elsewhere in the same program. GenProg and TSPRepair 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 [21]. GenProg minimizes code bloat post-facto; prior work has claimed that minimization reduces patches overfitting to the training tests [30]. 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 [55].

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* repair techniques share high-level goals (such as working reliably, applying at scale, and producing human-readable patches), 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, SemFix, and Debroy and Wong have successfully fixed bugs in legacy software. Although new projects appear to be increasingly adopting contracts [17], their penetration into existing systems and languages remains elusive. Few maintained contract implementations exist for widely-used languages such as C. Further, 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 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 [15, 30, 32, 34, 38, 41, 42, 52, 54, 56], some show that the resulting patches withstand red team attackers [42], and some consider the fraction of a set of bugs their technique can repair [25, 28, 38]. 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 [28], the relationship between operator choices and test execution parameters and success [29, 55], and human-rated patch acceptability [25] and maintainability [18]. However, these evaluations have not used a metric of correctness independent of patch construction, so little can be said about the correctness of the patches.

Our evaluation measures patch correctness in a rigorous manner independent 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.

This evaluation is a novel contribution to understanding *G&V* program repair. Stressing the importance of this work, concurrent research is starting to evaluate repair techniques in terms of overfitting [45, 52]. One evaluation [52] compares *relifix* to GenProg, evaluating the tools' tendency to introduce regression errors when constructing patches. That evaluation is a step toward the independent patch correctness evaluation we present here, but while our evaluation uses an independent test suite to measure the quality of the patch, this prior work uses only a subset of the test suite consisting of those tests that do not execute any of the lines that introduced the bug being repaired, thus ignoring specifically the regressions most likely to be introduced by a patch. Another concurrent evaluation is finding that using poor-quality test suites results in patches that overfit to those suites [45]. Our evaluation goes further, demonstrating that even using high-quality, high-coverage test suites still results in overfitting and finding other relationships between test suite properties and patch quality. Finally, human examinations of patches generated by Par and GenProg have measured acceptability [25] and maintainability [18]. While the human judgement 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

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 oracle 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.

told exactly where to look for them [40].

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, until the deadline, without penalty. 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 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.

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 or 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 [10]. 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. They are independent descriptions of the 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. Analogously, a repair tool can use the black-box tests to guide automated repair.

We use the two test suites as one measure of the functional *quality* of a patch. For example, if a human or tool used black-box tests in generating a patch, we then evaluate how well the patch performs on the held-out white-box test suite. If the patch passes all black-box tests 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 by providing them to GenProg, and evaluating resulting patches on the held-out black-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 are buggy as 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 use the dataset from Section 3 to evaluate *G&V* repair via a series of controlled experiments. Section 4.1 outlines the basic testing procedure we use to gather our data, and gives baseline numbers for successful patching in our dataset. Section 4.2 examines the overfitting we observed in *G&V* repair tools, and explores how various factors affect overfitting. Section 4.3 uses the human-written patches in our dataset to see if the repair tools we evaluate overfit more or less than novice student developers. Finally, Section 4.4 tests previously proposed avenues to combat overfitting.

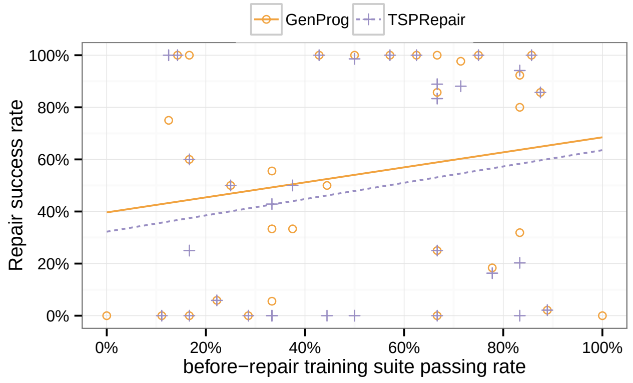
4.1 Testing Method

This section outlines the testing procedure we used to evaluate GenProg and TSPRepair.

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,

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.

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. For GenProg and TSPRepair, 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 evaluation suite over the patch and record the result.

Because AE is deterministic, some of our experiments do not apply to AE. AE cannot be run with different random seeds and so it produced less data that we could use in our statistical tests. As a consequence, while the results of all experiments for AE were consistent with those for GenProg and TSPRepair, we only report results for the AE experiments with respect to overfitting (Section 4.2).

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%), it is able to patch more scenarios (46.1% vs. 42.5%) and more distinct buggy program versions (61.9% vs. 57.1%) than TSPRepair. In contrast, AE is only able to fix 33.7% of buggy programs, having only one chance to fix each. Figure 2(b) shows how the number of black-box tests the un-patched buggy version fails affects the ability to produce a patch. 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). There is no significant relationship for AE. Based on these results, we conclude that GenProg and TSPRepair generate patches sufficiently often to enable further empirical experiments.

We state that we only report results with AE for overfitting, because there isn't enough data. So, I don't think I can say that AE generates patches sufficiently often to enable further empirical experiments without being inconsistent. – tks

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 linear.

4.2 Overfitting

Research Question 1: How often are the patches produced by *G&V* techniques overfit to the training test suite available during patch generation, and fail to generalize to a different test 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. To detect and measure this overfitting, we use the white-box suite as the held-out 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%). The median AE patch passes 62.5% of the evaluation suite (mean 61.9%).

We conclude that all tool-generated patches overfit to the training suite used in constructing the repair. For programs that are mostly correct to begin with, GenProg and TSPRepair *decrease* the correctness of the program under repair, as measured by the held-out suite.

Research Question 2: How does coverage of the training test suite affect patch overfitting?

In practice, test suites are typically incomplete. To measure how *G&V* techniques perform when given such realistic test suites, we use subsets of the black-box test suites as the training suite 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 how the coverage of the training test suite affects 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 the 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 tests suites, which are unfortunately common in practice, poses a risk of constructing patches that overfit to that test suite.

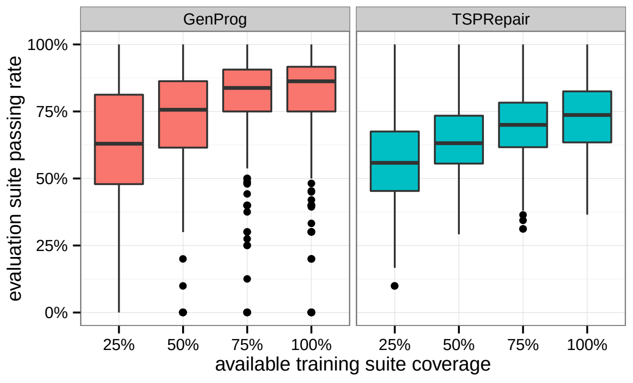


Figure 3: The coverage of the test suite GenProg and TSPRepair uses to repair the buggy program strongly correlated ($p < 0.001$) with the portion of the white-box tests the patched program passes.

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 affects a technique's ability to produce a patch. Next, we explore if it also affects the patch's overfitting.

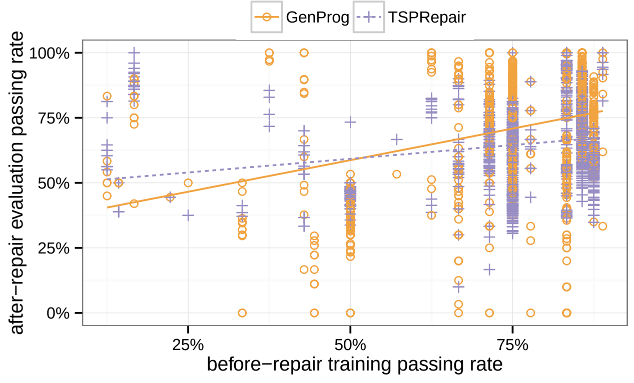
Figure 4 relates the quality of the generated patch, as measured by its performance on the held-out white-box tests, to the number of training black-box tests the original buggy program passes. Figure 4(a) shows that programs that passed more training tests before repair are more likely to pass the evaluation tests post-repair. Linear regression confirms the positive trend for both tools. However, Figure 4(b) shows that both GenProg and TSPRepair are more likely to break held-out white-box 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 with significance, $p < 0.001$ in both cases.

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

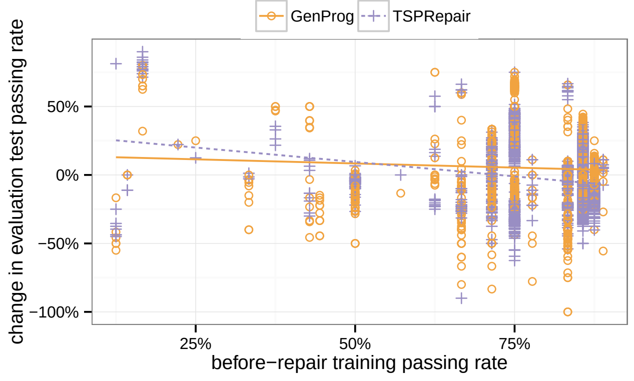
Research Question 4: How does the training test suite's provenance (automatically-generated vs. human-written) affect patches' overfitting?

Attempting to use low-coverage test suites to fix bugs can lead to low quality patches. Thus one may suggest using automatic test generation techniques to improve test suite coverage prior to repair. Here, we evaluate if automatically generated tests (generated with KLEE [10] 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 effect of test suite provenance on GenProg-generated patch overfitting. When GenProg repaired buggy programs using (all of) the black-box test suites as the training suite (Figure 5(a)), its patches did well on the white-box evaluation



(a)



(b)

Figure 4: While the portion of evaluation tests the patched program passes is significantly positively correlated ($p < 0.001$ for both tools) with the portion of training tests the un-patched version passes (a), both tools are more likely to break previously working evaluation tests for programs that pass more training tests before repair (b); the correlation between before-repair training suite pass rate and evaluation tests fixed is significantly negative ($p < 0.001$).

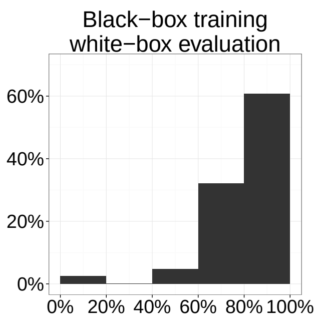
tests. However, the same was not true when GenProg repaired using (all of) the white-box test suite as the training suite, and the black-box tests as the held-out suite (Figure 5(c)). In the latter case, GenProg overfit significantly to the white-box tests. Figure 5(e) directly compares the two provenance methods. A two-sample test rejects the null hypothesis of similar distributions of the two samples, and instead suggests that the black-box patches pass significantly 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 rejects the null hypothesis with significance ($p < 0.001$), and a Cliff's Delta test reports a similar large-magnitude effect (magnitude > 0.5).

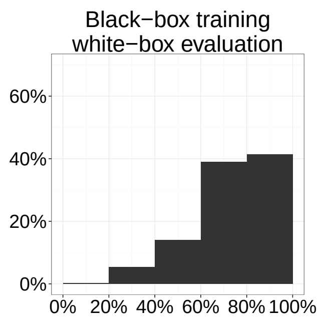
We conclude that test suite provenance plays an important role in GenProg-generated patch quality. Not all full-coverage test suites are created equal, and some are more suited for automated repair than others.

GenProg

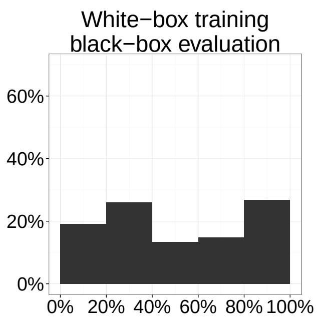
TSPRepair



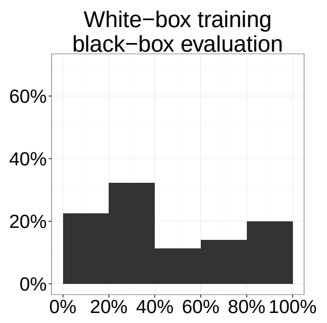
(a) White-box passing rate of GenProg patches generated with black-box tests.



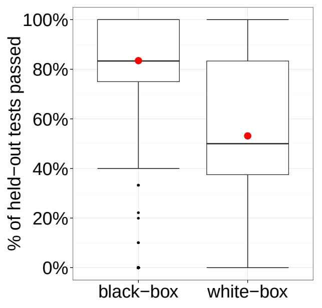
(b) White-box passing rate of TSPRepair patches generated with black-box tests.



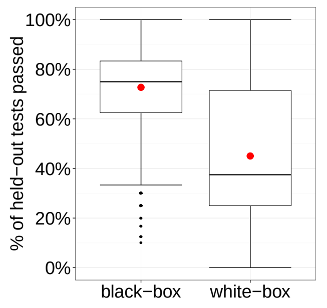
(c) Black-box passing rate of GenProg patches generated with white-box tests.



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



(e) Tests GenProg used to guide repair. The line shows the median, and the point the mean.

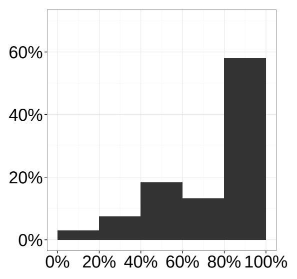


(f) Tests TSPRepair used to guide repair.

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 search for patches. (c) 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. For both tools, Wilcoxon signed-rank tests detected a significant difference, $p < 0.001$ with a large Cliff's Delta in both cases.

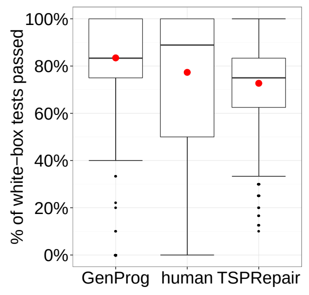
4.3 Do tools outperform novice developers?

One of the advantages of our dataset is that every program has a human fix associated with it, corresponding to the final submission for each student. The students who produced the buggy programs in our dataset are faced with a challenge similar those presented



(a)

White-box passing rate of human-written patches who used black-box tests.



(b)

Direct comparison. The line shows the median, and the point the mean.

Figure 6: When evaluated on the held-out white-box tests, patches generated using black-box tests are of slightly better quality than human-written patches (b), on average. (c) compares the distributions explicitly: the human-written patches show higher variance, and while the median of the human (line) is higher, the mean (point) is lower. TSPRepair performs worse, with a lower median and mean than human-written patches. The Wilcoxon signed rank test with an alternative hypothesis that the mean of the GenProg data is higher than the mean of the human data rejects the null hypothesis ($p < 0.001$), while the same test for the TSPRepair data cannot reject the null hypothesis.

to our repair tools. They write and submit code, gain information about how many tests their code passes and fails, make a change, 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 results of the automated repair tools to the results of human 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 less than novice developer 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 human, GenProg, 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 finds a difference between the GenProg-generated and human-written patches (Figure 6(b)), and rejects the null hypothesis (with high significance, $p < 0.001$) in favor of an alternative that the average of the GenProg patch performance is higher, indicating that, in fact, GenProg slightly outperforms students. However, the improvement is only slight: 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 variability in quality than human-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 human-generated patches and Genprog-produced patches. While there is a visual difference in Figure 6(b) between human-written and TSPRepair-generated patches, the Wilcoxon signed rank test reports no significant difference between the samples. As with GenProg, TSPRepair-generated patches demonstrate significantly less variability in quality than human-written patches.

Comparing automatically generated patches to human-written patches might seem unfair since repair tools can only access 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, 33] 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 [57], to reduce code bloat. TSPRepair does not perform minimization, because the produced patch is only ever a single edit. **[[AE?]]** Intuitively, a small change to a program is less likely to encode special behavior that handles just the training tests in a separate way. Instead, a small change is more likely to encode a generalization of the requirements [56]. 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 makes the overfitting behavior even worse.

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 has no effect on how much GenProg-generated patches overfit.

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

Randomized algorithms in repair tools 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, if each patch in a group encodes most of the desired behavior, even if each one overfits on some subset of that behavior, as long as the overlap in the patches' overfitting is small, the group voting on the behavior may outperform each individual patch. This n-version patch may provide an avenue to mitigate overfitting. Human-written code typically lacks sufficient diversity [26] to enable n-version programming [13], 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 $\mathcal{P}_p^1 \ldots \mathcal{P}_p^n$. Next, we create a new program, $\mathcal{P}_n$, that on input i, runs each of $\mathcal{P}_p^1 \ldots \mathcal{P}_p^n$ on i, and returns the output frequently returned by output by those program. If two or more return values tie, $\mathcal{P}_n$ returns one at random.

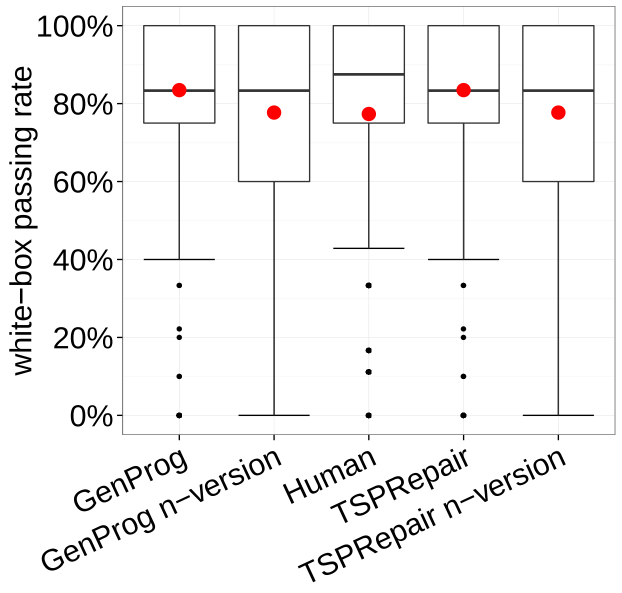


Figure 7: Tool-generated patches and n-version programs made up of those patches perform worse than humans-written patches, on average. N-version GenProg programs underperform even the individual GenProg patches, and n-version TSPRepair programs perform negligibly worse than individual TSPRepair patches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

Figure 7 shows that n-version patches constructed from either tool's output do not perform statistically significantly better than either individual patches or human-written patches. The only exception are white-box trained GenProg patches. For this case, a Wilcoxon rank-sum test detects a significant difference between the individual patches and n-version patches ($p < 0.001$), but the Cliff's effect size is *small*.

We conclude that when test suites enable a tool to produce quality patches, 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

Test suite provenance had the largest effect on the quality of resulting automatically-generated patches. This section describes one instance of a buggy student program and two patches that GenProg produced for it to highlight the ways that the automatically-generated white-box tests can lead the patch search process astray. 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 test suites, both black- and white-box, used to evaluate both the student submissions and GenProg patches. The students had access to neither suite while completing the assignment, beyond their program's performance on the black-box tests; for the purposes of this discussion, GenProg had access to only the white-box suite. We use the black-box suite to evaluate final patch quality.

One of the student's (non-final) submissions to the homework system 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 any answer for the fifth black-box and second white-box tests, where `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 have been removed, such that this version 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. To summarize, this code is correct for inputs in which: `n1` is the median and `n3 < n2`, `n2` is the median while being greater than `n1`, or `n3` is the median and `n2 <= 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.

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 different random seeds can lead to different patches for the same bug. For example, for this buggy program, GenProg also produced a patch that leads to:

```
 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. Neither test suite adequately rooted out this particular logic bug. However, this patch inserted a copy of the `return n3` into the `else` block of the last set of conditions, which seek to determine whether `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 a median.

Before submitting the final version, 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 human-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.

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`.

6. THREATS TO VALIDITY

Our experiments may not generalize. We only experiment with GenProg and TSPRepair **[[AE!!!]]**, two of several *G&V* repair techniques [55], and our results may not extend to other automatic program repair mechanisms. Our subject programs are small student programs, with fairly small test suites. While these are experimentally interesting in that they provide a very large set for conducting controlled trials, the programs' small sizes may affect the ability to find diverse patches. We ran 20 seeds per repair effort, a relatively small number by the standards of metahueristic search algorithms. More attempts may have revealed more solutions. Finally, we used the recommended GenProg parameter set defined in previous work [29]; a full parameter sweep is outside the scope of this investigation.

We release our dataset, including all the buggy versions, human-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, attempted to control 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

Empirical studies of fixes of real bugs in open-source projects can help tool designers select change operators and search strategies [24, 58].

Some automated repair techniques focus on a particular defect class. Certain classes of security vulnerabilities, such as buffer overruns, have received especial attention [49, 51]. Templated repair can address unsafe integer use in C programs [15]. AFix uses static analysis to address single-variable atomicity violations [23]. Lin and Kulkarni address deadlock and livelock defects [31], and Grail [32] attempts to fix general concurrency errors using a context-aware synthesis. The ARMOR tool [11] replaces buggy library calls with different calls that achieve the same behavior. Alkhalaf *et al.* address input validation and sanitization [3]. *relifix* uses a set of templates mined from regression fixes to automatically patch regression bugs. By contrast, GenProg is more general and has addressed security and non-security bugs.

User-provided code contracts, or other forms of invariants, can help to *synthesize* correct-by-construction patches, e.g., via AutoFix-E [41, 54] (for Eiffel code) and SemFix [38] (for C). DirectFix [34] aims specifically to synthesize minimal patches specifically to less prone to overfitting, but only works for programs with a subset of language features, and has only been tested on small (median hundreds LoC) programs. These techniques have the benefit of correctness proofs, but require the additional human burden of providing the contracts, and inherit the limitations associated with the underlying proof system. Synthesis techniques can also construct new features from examples [14, 20], rather than address existing bugs.

GenProg [28, 30, 56] is representative of *G&V* approaches, as we define in Section 2.1. 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, 16, 25, 34, 38, 39, 42, 55]). Section 2.3 has already discussed previous evaluations of *G&V* techniques. Monperrus [37] recently discussed the challenges of experimentally comparing program repair techniques. For example, the selection of test subjects (defects) can introduce evaluation bias. Our evaluation focuses precisely on the limits and potential of repair techniques on a large dataset of defects, controls for a variety of potential influencers, and accounts for statistical analysis of our results, addressing some of Monperrus' concerns [37].

Genetic programming tends to produce extraneous code that does not contribute to the fitness of the solution [21, 50]. GenProg attempts to mitigate this through solution minimization. Overfitting is also a well-studied problem in machine learning [36]. Our experiments on patch minimization and overfitting suggest that there the two are unrelated, which is consistent with other results in machine learning [47]. To the best of our knowledge, ours is the first consideration of this relationship in the domain of program repair.

Search-based software engineering [22] adapts search methods, such as genetic programming, to software engineering tasks, such as developing test suites [35, 53], finding safety violations [2], refactoring [48], 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 [13] combines multiple different programs trying to solve the same problem in the interest of achieving resiliency and correctness through redundancy. It has been shown to work poorly with human-written systems because the errors humans make do not appear to be independent [26]. Using n-versions with GenProg-generated patches did improve on individual patches produced using low-quality test cases, but still 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 significant practical traction, we must augment feasibility demonstrations with qualitative evaluations that address the techniques' quality and applicability. In this paper, we systematically evaluated the factors affecting the output quality of GenProg and TSPRepair **[[AE???]]**, which we believe to be representative of *G&V* techniques, through a controlled and comprehensive evaluation on a large set of student-written programs with naturally occurring bugs and human-written patches. Based on our findings, we conclude that the remaining research challenges include:

Repair techniques must go beyond testing on the training data to characterize functional correctness. GenProg produced a patch for more than half (61.9%), and TSPRepair produced a patch for almost a third (31.2%), **[[AE?]]** of the bugs in our dataset. 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 human programmers (students) *also* overfit to the provided test cases. When using requirements-based tests, GenProg overfit less than the humans. These results highlight both the significant promise of automatic repair and the fact that more work is needed to improve their output quality.

To correct for this, we propose that future evaluations of *G&V* repair tools **withhold** some proportion of tests from the vantage point of their repair tool, at least one of which shares code-under-test with the tests which expose 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 in the codebase.

Automatic repair should be used in appropriate contexts. Both test suite coverage and input program quality affected the quality of the automatic patches. Higher coverage test suites 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 mitigate these tensions.

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 impact on patch quality. Automatic test-input generation techniques *should* fit naturally into a tool chain 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. These 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. Despite their relatively lower quality, the patches generated using automatically generated tests demonstrated sufficient functional diversity to improve on the patched programs via plurality voting. The patches generated on the human-written tests, however, showed minimal such diversity. 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 human 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.

Acknowledgements

Prem Devanbu and Ming Xiao were instrumental in the creation of an earlier version of the student programs dataset and early GenProg experiments [9].

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] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] P. Liu, O. Tripp, and C. Zhang. Grail: context-aware fixing of concurrency bugs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 318–329. ACM, 2014.

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

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

[48] 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.

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

[50] 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.

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

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

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