Evolution vs. Intelligent Design in Program Patching

Yuriy Brun
School of Computer Science
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

Earl Barr
Dept. of Computer Science
University College London
London, UK
e.barr@ucl.ac.uk

Ming Xiao
Dept. of Computer Science
University of California
Davis, CA, USA
minxiao@ucdavis.edu

Claire Le Goues
Inst. for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
legoues@cs.cmu.edu

P. Devanbu
Dept. of Computer Science
University of California
Davis, CA, USA
devanbu@ucdavis.edu

ABSTRACT

While fixing bugs requires significant manual effort, recent research has shown that genetic programming (GP) can be used to search through a space of programs to automatically find candidate bug-fixing patches. Given a program, and a set of test cases (some of which fail), a GP-based repair technique evolves a patch or a patched program using program mutation and selection operators. We evaluate GenProg, a well-known GP-based patch generator, using a large, diverse dataset of over a thousand simple (both buggy and correct) student-written homework programs, using two different test sets: a white-box test set constructed to achieve edge coverage on an oracle program, and a black-box test set developed to exercise the desired specification. We find that GenProg often succeeds at finding a patch that will cause student programs to pass supplied white-box test cases; however, that the solution quite often overfits to the supplied tests and doesn't pass all the black-box tests. In contrast, when students patch their own buggy programs, these patches tend to pass the black-box tests as well. We also find that the GenProg-generated patches lack enough diversity to benefit from a kind of bagging, in which a plurality vote over a population of GP-generated patches outperforms a randomly chosen individual patch. We report these results and additional relationships between GenProg's success and the size and complexity of the manual and automatic patches.

1. INTRODUCTION

Software permeates many aspects of our lives, transforming, for example, healthcare, business, and communication. Ensuring that software is of high quality has thus become vitally important. Unfortunately, debugging and improving software quality is quite expensive. A 2013 study estimates the global cost of debugging as $312 billion and that developers spend half their time debugging [10]. In 2008, on average, it took human developers 28 days to resolve a security bug [37] and new bugs are reported faster than developers can handle them [3].

Recently developed automated bug-fixing techniques [12,13,15, 18,23,24,26,27,31,33,39,41,43,44] have the potential to reduce costs and improve software quality. Recent studies speak to this potential while raising new questions about the acceptability of automatically generated patches to humans. The GenProg tool [44] quickly and cheaply generated patches for 55 out of 105 historical bugs in open-source C programs [26], while the PAR tool showed comparable results on 119 bugs in Java programs [24]. At the same time, the latter study showed that patches generated by PAR and GenProg are considered acceptable by humans 23% (27 out of 119) and 13% (16 out of 119) of the time, respectively. Meanwhile, machine-patched code is slightly less maintainable than human-patched code, though the trend is mitigated entirely by the use of automatically-synthesized documentation [19]. These modest but encouraging successes suggest great promise for automated bug fixing while making evident that significant work remains to understand and improve the quality of the patches such techniques produce.

GenProg employs a *genetic programming* (GP) metaheuristic to search the space of code patches for a particular bug in a program. First, GenProg locates potentially defective code regions using existing fault localization techniques. Next, GenProg generates a population of potential patches for the buggy program. The patches can delete defective code, or replace or augment the defective code with code extracted from other regions of the defective program. GenProg employs genetic mutation, breeding, and selection operators, using test-passing behavior as fitness, over this population. As discussed above, this approach has been quite successful.

In general, however, search-based problem-solving approaches run the risk of overfitting solutions to the objective function [32] (a test suite, in this case). Consider organisms that develop highly specific adaptations to fit their environments (e.g., the koala to eucalyptus groves). Such creatures can become vulnerable if environmental conditions shift. By analogy, it is possible that patches generated by an evolutionary fitness-seeking process will be specific to the given test cases, and not fare as well with different tests. By contrast, an intelligently designed fix, constructed by a human who holistically considers the requirements (rather than just a few known target test cases), might be robust enough to pass a larger variety of tests. Our goal here is to evaluate this phenomenon, and seek mitigations as applicable.

Our experimental setting is a large, diverse dataset of over a thousand failing, student-written programs, all submitted as homework in a freshman programming class, and all with student-written, bug-

WOODSTOCK '97 El Paso, Texas USA

fixing patches. While admittedly, our dataset is homogenous in that all our programs, bugs, and fixes are written by beginner programmers, it is rich in other ways, such as the availability of human-written fixes, and many different implementations, both buggy and correct, for the same specification. The programs' small size allows us to better understand how GenProg succeeds and fails, which would be more difficult on large, complex programs. Further, and critically, the conclusions we draw about GenProg's failures on these simple programs are very likely to generalize to larger, more complex programs: If GenProg cannot effectively patch simple bugs in simple programs, it is unlikely to be successful with complex bugs.

We evaluate GenProg in this setting using two different test suites, one for training and one for testing. We developed these suites using different approaches: We constructed the training suite using a white-box coverage approach; whenever feasible, we used an automated coverage tool, KLEE [11], to generate these suites. We constructed the test suite using a black-box coverage approach, using the homework descriptions as requirements. We used, as a starting point, buggy student programs that failed to pass all the tests in the training suite, and evaluated whether (1) GenProg could patch the buggy programs to pass *all* the tests in the training suite; (2) the patched programs would also pass all the tests in the test suite; (3) whether the population of patched programs arising from each student buggy program contained, within itself, enough diversity to pass all the tests in the test suite, and (4) if minimizing the patches affected their quality.

We offer the following four contributions:

1. A large-scale evaluation of automatically generated bug patches. This evaluation uses GenProg to fix over a thousand student-written, naturally buggy programs, and compares the generated patches to human-written ones.
2. An evaluation of the tendency of evolved patches to over adapt to provided test cases, and thus show brittleness in the face of new test cases.
3. An exploration of whether a population of automatically patched programs offers useful diversity that can be used to improve automatic patching techniques.
4. A benchmark of programs with bugs that were written by real humans, albeit beginner programmers, and human-written patches for those bugs. This benchmark can be invaluable in evaluating and improving future automatic patching techniques.

The rest of this paper is structured as follows. Section 2 discusses the background of evaluation of automated patch generation techniques. Section 3 outlines our experimental design. Section 4 presents our dataset. Section 5 details our experimental results and Section 6 qualitatively demonstrates our findings on one example datapoint. Finally, Section 7 acknowledges threats to the validity of our results, Section 8 places our work in the context of related research, and Section 9 summarizes our contributions.

2. BACKGROUND

Automatically patching bugs [5,42] has gained significant attention since the introduction of GenProg [44] and ClearView [33]. The primary difference between these approaches is that GenProg uses mutation to generate many candidate patches, whereas ClearView mines invariants of good executions and then generates candidates that enforce those invariants on the bad executions. Both techniques then check the candidates via testing. Other automated patching techniques use constraints to build a single patch that is correct by construction through the use of formal verifications or programmer-provided contracts or specifications (e.g. [23,41]). Since then, each of these approaches has generated new patching techniques [12, 13, 15, 18, 23, 24, 26, 27, 31, 39, 41, 43].

In this paper, we focus on the automated techniques that generate many candidate patches or repaired programs and then validate them using tests. This paradigm includes GenProg and Clearview, but typically not those techniques that provide correctness guarantees with respect to a user-provided specification. These techniques are often grounded in search-based software engineering [22], searching through a space of possible programs to find one that satisfies a set of conditions. GenProg in particular uses GP to search through candidate patches that selectively mutate the buggy program. GenProg takes as input a buggy program and a set of tests, some of which fail. Then, GenProg generates a population of small random patches. After computing a fitness score for each mutant — the number of tests the patch causes the buggy program to pass — GenProg keeps only the fittest mutants, recombining and editing them using GP mechanisms, until either a mutant causes the program to pass all the tests or a preset time or resource limit is achieved. Because GP is a random search technique, GenProg is typically run multiple times, on different random seeds, in an effort to repair a single bug.

There are three key challenges to GenProg's goal. First, there are many places in the program that may be mutated. GenProg tackles this challenge by using existing fault localization techniques to prioritize mutating program constructs (such as statements in C) that are executed exclusively by the failing test case(s) over those that are also executed by the passing executions (under the assumption that lines that execute on faulty tests are more likely to contain the bug). Second, there are many ways in which code may be mutated. GenProg tackles this challenge by limiting the mutations to line deletions or copying lines of code from elsewhere in the same program (under the assumption that if your code has a bug, there is a good chance it had the same bug elsewhere, and you already fixed it there). Third, metaheuristic search strategies, GP in particular, to code is prone code bloat [20]. That is, the patches GenProg produces, while causing the program to pass all the supplied test cases, is likely to also contain extra code that affects only the behavior not tested by the supplied test cases. GenProg tackles this challenge by using delta debugging [45] to minimize the amount of code affected by the patch that passes all tests.

There have been several evaluations of GenProg's effectiveness. The first, on 9 bugs that occurred during real development in 9 open-source programs of 1K to 22K LoC (plus one toy example buggy program), showed that an average GenProg run succeeded at finding a patch 59% of the time, and GenProg was able to fix all 9 bugs [44]. A more recent study evaluated the technique on 105 bugs taken systematically from the repositories of 8 open-source C programs of 62K to 2.8M LoC, and showed that GenProg patched 52% (55 out of the 105) of the failures [26]. An independent evaluation of the technique on 119 bugs from 6 open-source Java programs of 28K to 180K LoC showed that 13% (16 out of the 119) of GenProg's bug patches were considered acceptable by humans [24]. Another recent study [19], found that GenProg patches are slightly less maintainable than human-written patches, but that automatically-synthesized documentation can reverse this trend entirely.

These evaluations have demonstrated that for a fairly small number of bugs in medium-sized programs, GenProg can generate patches; however, humans do not often consider those patches acceptable on their own. In contrast, our goal is to evaluate GenProg on a much larger set of bugs (albeit we were forced to significantly reduce the size of the programs to achieve a large-enough set of bugs) and, particularly, to explore whether GenProg patches overfit to the supplied tests, which we believe could be the reason humans do not find many patches acceptable.

GenProg, and many other related bug patching 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. In our work, we tried to be consistent with the guidelines for evaluating randomized algorithms [4] to enhance the credibility of our findings. Specifically, we used a large sample of over a thousand buggy student programs, attempted to control for a variety of potential influencers in our experiments, and used fixed-effects regression models (Section 5.1) and two sample tests (Section 5.3) along with false-discovery rate correction to lend statistical support to our findings.

3. RESEARCH QUESTIONS AND EXPERIMENTAL DESIGN

We are interested in examining how well GenProg works on patching bugs. This section describes our experimental setup and hypotheses, and Section 4 will describe our dataset. While we design these experiments to be agnostic of the dataset, and thus more easily repeatable on other datasets, we assume that for each subject program specification we have (1) multiple buggy solutions, (2) a human-written patch (some successful, others perhaps not) for each buggy solution, and (3) a single reference oracle solution that satisfies the specification. The bugginess of the buggy solutions may vary greatly. In our experiments, we allowed the buggy solutions to pass anywhere from only one to all-but-one of the program tests we describe next.

3.1 Developing Test Suites

For each subject program, we first develop two suites of tests: a white-box and a black-box suite. The *white-box test suite* is based on an oracle program whose output is, by definition, the correct output the program should exhibit. The white-box test suite achieves edge coverage (also called branch coverage) on the oracle. Whenever possible, we create the white-box test suite using KLEE, a symbolic execution tool that automatically generates tests that achieve high coverage [11]. When KLEE fails to find a covering test suite (typically because of the lack of a suitable constraint solver), we construct a test suite manually to achieve edge coverage on the oracle.

The *black-box test suite* is based only on the program specification, or description. We construct this test suite manually, using equivalence partitioning: We carefully separate the input space into several equivalent partitions, based on the specification, and select one input from each category. For example, for a program that computes the median of three numbers, we would provide tests with the median as the first, second, and third input, and also tests with two equal inputs, and all three equal inputs.

The white-box and black-box test suites are developed independently. They provide two separate descriptions of the desired program behavior: one to provide a target to allow GenProg to evolve a patch, and the other to provide a way to challenge and evaluate the evolved patch.

While the white-box and black-box suites are the same for all buggy versions of each subject program, we also develop a smaller *training set* for each buggy version. The training set is a subset of that subject program's white-box suite, of size either 25%, 50%, 75%, or 100% of the cardinality of the white-box suite. The training suite is such that the buggy program passes at least $\frac{1}{2}$ of the tests in the suite, but also fails at least one test. The goal of the training set is to provide a larger diversity of challenge for GenProg: Patching a program that conforms to a smaller number of tests is easier than to a larger number. Thus, the training set allows us to evaluate GenProg's ability on varying levels of challenge. (As we describe in Section 4, selecting this training suite can also be helpful in finding buggy versions in a revision history of development.)

3.2 GenProg Effectiveness

First, given a program, a suite of tests that program passes, and a suite of tests that program fails, we set out to measure how often GenProg finds a patched program that passes all the supplied tests.

Research Question 1: How often does GenProg find a solution that patches a buggy program to pass all the training tests? How does the rate of success depend on the severity of the flaws of the buggy program?

For each of the buggy programs, we run GenProg, giving it access to the training test suite. Since on each seed, GenProg either produces a patch, or times out, and since we plan to run our experiments on a large dataset, to optimally utilize the computing resources (e.g., to not spend a lot of computing trying many seeds on a version that GenProg will always time out on), we developed a progressive patching strategy that attempted anywhere between 5 and 55 runs per buggy version. We first run each GenProg on 5 seeds. If GenProg is unable to generate a patch on these runs, we mark that version as patched by GenProg 0% of the time. However, for each successful patching, we allow GenProg five more tries. We continue this process until either we run out of tries, or we reach 10 successful runs (and thus 10, not necessarily distinct patches). This allows us to spend less time focusing on unsuccessful runs, and achieve more precision in GenProg's success rates for the buggy versions that are neither trivial nor nearly impossible for GenProg. Having said this, in our experience, with extremely rare exceptions, GenProg either produced a successful patch on nearly every run, or was unable to produce a patch on every run.

We compute two measures of success: (1) the fraction of the buggy versions for which GenProg was able to produce at least one patch, and (2) the fraction of the GenProg runs for each buggy version that succeeded in producing a patch. Further, we examine how the level of difficulty — fraction of the training suite of tests the buggy version fails — affects these two success measures.

3.3 GenProg Solution Specificity

Mutation and natural selection mechanisms in evolution drive populations of organisms toward high fitness within their ecosystem. Analogously, GenProg uses a GP metaheuristics that drives its patch evolution toward passing all the *given* test cases. The question then naturally arises, "Does this evolutionary process produce patches that are overly specific to the given test cases?"

Research Question 2: How often are the patches produced by GenProg *over-specific* to the tests made available to GenProg, and fail to generalize to a larger set of tests, and to the program specification? Do human-written patches generalize better?

Using the training suite and the black-box suite, we create two independent evaluation criteria of the patches. GenProg attempts to patch the buggy program having access only to the training suite.

Then, we evaluate the patch GenProg generates on the black-box suite to determine how well the patch satisfies the program specification.

Recall that for some buggy programs, the training suite is equal to the entire white-box suite, while for others, GenProg's challenge of finding a program that passes all the black-box tests is greater, as it is given fewer tests that define the program's behavior. As incomplete test suites are typical in real life, this represents a realistic situation. It further allows us to explore the relationship between the amount of guidance given to GenProg, in terms of completeness of the test suite, and the generality of GenProg's patches. This leads us to answer the following question:

> **Research Question 3:** How does the generality of the GenProg-generated patch depend on the fraction of the white-box test suite provided to GenProg?

3.4 GenProg Solution Diversity

The previous experiment explores if GenProg-generated patches overfit to the training suite uses to evolve the patch. Even if the patches do overfit, there is a silver lining: Since GenProg uses randomness in its evolutionary approach, it is possible that a group of patches better represents the desired program behavior than each patch individually. In other words, while each of multiple patches may overfit, they each may perform properly on some large subset of the desired behavior, and only overfit — perform improperly — on a relatively smaller subset of that behavior. Further, the for any two patches, the subsets of the behavior for which they misbehave may differ. If true, this would allow using a collection of patches together, running all of them on each input and then having them vote on the correct result. If the overfitting is diverse enough, this n-version patch would outperform individual patches.

N-version programming [14] suffers when the versions lack diversity, as is typically the case with human-written code [25]. Knight and Levison found that independently developing several different versions of the same program, the errors programmers made did *not* appear to be independent. In our scenario, the patches are evolved automatically by an essentially randomized algorithm, presenting a reasonable setting to expect that the evolved fixes *might* be relatively independent. It is our goal to evaluate the diversity of the GenProg-generated patches for this purpose.

The procedure for creating the n-version program $\mathcal{P}_n$ is as follows: For each buggy program version $\mathcal{P}_b$, which partially passes a training suite of tests, run GenProg on $\mathcal{P}_b$ with $\mathcal{P}_b$'s training suite of tests to generate $n \geq 3$, distinct patched program versions $\mathcal{P}_p^1 \dots \mathcal{P}_p^n$. Then create a new program, $\mathcal{P}_n$, that on the set of inputs i, runs each of $\mathcal{P}_p^1 \dots \mathcal{P}_p^n$ on i, and returns the most frequently returned output by those programs. If two or more return values are equally most-frequent, return one at random.

We then evaluate $\mathcal{P}_n$ on the black-box test suite, and compare its success to the success of each buggy version, each individual GenProg-generated fix, and the human-written patch. This allows us to answer the following research question:

> **Research Question 4:** Does the population of GenProg-generated patches (on different random seeds) have enough diversity to benefit from a kind of bagging, in which a plurality vote outperforms a randomly chosen individual? How does the success of the plurality vote compare to that of the original buggy version, the individual GenProg-generated fixes, and the human-written patches?

program	**LoC**	**tests**		**computation**
		white-box	**black-box**	
checksum	13	10	6	checksum of a string
digits	15	10	6	digits of a number
grade	19	9	9	grade from numeric score
median	24	6	7	median of 3 numbers
smallest	20	8	8	minimum of 4 numbers
syllables	23	10	6	count syllables in a word

Figure 1: Summary of our dataset's subject programs and test cases.

3.5 GenProg Patch Minimization Effectiveness

Instead of using diversity to *mask* solutions that overfit, it may be possible to eliminate the overfitting from individual patches. GenProg offers one mechanism that may help to do so: minimization. While GenProg uses patch minimization, via delta debugging [45], to reduce code bloat, minimizing the patch may also reduce overfitting. Intuitively, a small change to a program is less likely to encode special behavior that handles just the supplied tests in a separate way. Instead, a small change is more likely to encode a generalization of the requirements.

On the other hand, it also possible that minimizing the patch makes it overfit even more strongly to the supplied tests because in some sense, minimizing removes any code that isn't specific to the supplied tests the patch makes pass.

> **Research Question 5:** Does minimizing the GenProg-generated patches affect the specificity of the patches?

To answer this research question, we measured the effect of minimization of the patches on the black-box suite passing rates. An increase in the passing rate means the minimization decreased overfitting, whereas a decrease in the passing rate means the opposite. We also compared the minimized patches against the human-written patches.

4. THE DATASET: PROGRAMS AND TESTS

This section describes the programs we used to evaluate GenProg. We first describe the programs in general, and then how we obtained and selected buggy versions of those programs.

4.1 The Programs

Our dataset is drawn from an introductory C programming class (ECS 30, at UC Davis) with an enrollment of about 200 students. There are six different programming assignments (see Figure 1). Each assignment consists of the students, individually, writing 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 that 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.

For submitting each assignment, each student has access to a unique git repository; to submit the assignments, the students push their changes into these repositories via `git push`. The students may submit as many times as they desire, until the deadline. There are no penalties for multiple submissions.

On every submission, a system called GRADEBOT runs the student program against a collection of test cases. The student pro-

program	training suite size	buggy versions	attempts
checksum	0.25	26	989
	0.50	25	1,204
	0.75	24	1,067
	1.00	20	121
digits	0.25	85	1,844
	0.50	76	2,831
	0.75	71	3,266
	1.00	70	454
grade	0.25	81	2,129
	0.50	79	2,220
	0.75	75	2,862
	1.00	66	302
median	0.25	62	5,102
	0.50	62	3,972
	0.75	62	4,422
	1.00	30	265
smallest	0.25	55	1,643
	0.50	55	1,841
	0.75	20	1,635
	1.00	19	166
syllables	0.25	38	2,123
	0.50	37	1,624
	0.75	42	1,344
	1.00	32	153
total		1,212	43,579

Figure 2: Summary of the buggy program versions we found for every size of the training suite (as a fraction of the white-box suite size). Each buggy version was the latest submission that passed at least half, but not all of the training suite. The *attempts* column lists how many attempts were made in total to patch buggy versions at that level.

gram output is compared against the output of an instructor-written oracle. (The comparison to oracle's output is insensitive to whitespace and capitalization). GRADEBOT tells the student the total number of tests run and how many of those tests the submission passed, but no other information about the tests. The homework grade is proportional to the fraction of tests the latest submission (before the deadline) passes.

GRADEBOT requires students to pass all tests to receive full credit. Students can, at all times, query the oracle for the correct output on a student-supplied input, but students do *not* know the test cases used by the GRADEBOT. When students find that their submission fails a test, they are forced to consider their program carefully, in light of the requirements, to see what they are missing. Thus, the students' patches and repairs are driven more by the requirements, rather than by specific tests. Therefore, the student-written solutions provide intelligently designed programs to compare against the patches produced by GenProg.

Finally, for each assignment, we also had an instructor-written oracle implementation that satisfied the requirements fully.

4.2 Test Suites

Following the procedure described in subsection 3.1, we created a white-box and a black-box suite of tests for each program. Figure 1 describes the sizes of these suites.

subsection 3.1) also described each buggy version of the program having a training suite. For our dataset, we created the training suite before we identified the buggy program; in fact, we used the training suite to identify the buggy programs. We selected the training suites by choosing randomly, with replacement, suites of size 25%, 50%, and 75% of the entire white-box test suite. For each size, we created at least ten subsamples. These subsamples, together with the white-box suites themselves, formed the set of training suites.

The reason for picking ten subsamples is to average out random variations. Since there are many ways of chose four test cases out of eight, we ensured that we used a representative set of such subsets by making ten subsamples.

4.3 Buggy Versions

Because the homework submission process is handled via a git repository, this process creates a careful history of multiple versions of the code each student wrote as they were developing it. Inevitably, some versions are buggy, and many, though not all, of the final versions are correct.

We used the training suites (subsamples of the white-box suite described in subsection 4.2) to find the buggy versions. For each student submission repository, we selected randomly a single 25% training suite, a single 50% training suite, a single 75% training suite, and the full (100%) white-box suite. For each of these four suites, we searched the repository for the latest version of the program that passed at least half of the suite, but did not pass all of them. Each such version we found represented a buggy program that we use in our experiments.

Not every repository contained such a buggy program. For example, a student could submit a correct version without ever submitting a buggy one, or could first submit a completely wrong program that failed all the tests but then write the correct solution without intermediate submissions. The *count* column in Figure 2 shows the number of buggy programs we found at each subsampling level.

This procedure allowed us to find buggy programs of different levels of bugginess. Our goals were to get a diversity of bugginess, and also not to enforce a strict number of failing tests per version. Instead, we wanted a more naturally occurring sample. Thus, in our search, we imposed only a lower-bound threshold on the number of tests that had to pass. For example, the versions we found with the training suite equal to 25% of the white-box suite were required to only pass half of that suite, or 12.5% of the white-box suite. Some of these passed more, although every one was required to fail at least one test. Meanwhile, the versions we found with the training suite that equaled the white-box suite had a higher lower-bound threshold: They were required to pass at least half of all the white-box tests. We feel that this procedure gave us a representative sample of the buggy versions present in the student submission repositories.

We will make the programs, version histories, and test cases publicly available, after salting and MD5-hashing student names for privacy.

5. EXPERIMENTAL RESULTS

In this section, we present the results of our experiments, and answer the research questions from Section 3.

5.1 GenProg Effectiveness

We tested GenProg's effectiveness by fixing the buggy versions of six programs, as described in Section 4. Figure 2 summarizes how many buggy program versions of each program we found at every size of the training suite (25%, 50%, 75%, and 100%), and

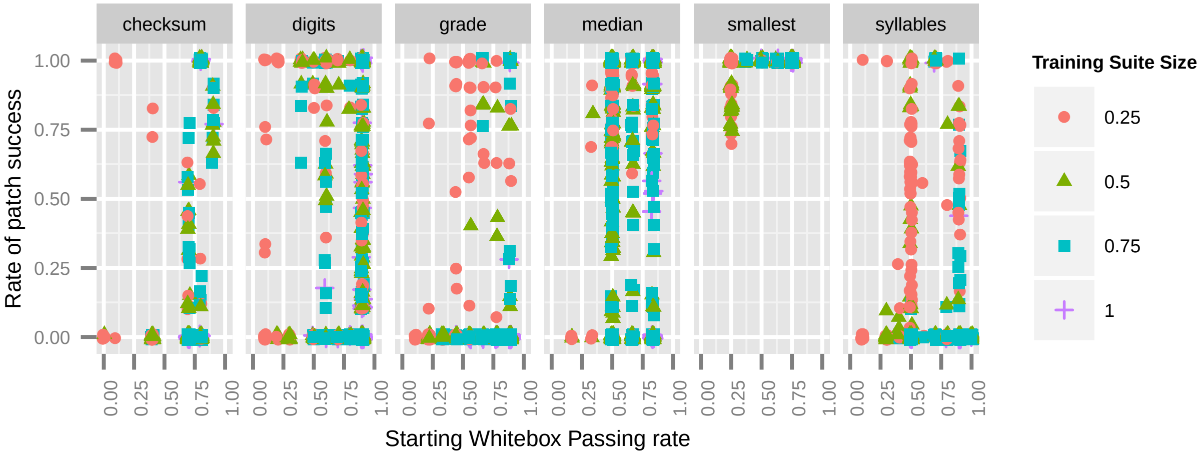


Figure 3: GenProg's success rate of patching a buggy program varies with the training suite.

the number of attempts we made at patching these program versions, with the different training suites, following the procedure described in Section 3.2. Overall, we attempted patching 1,212 versions, in 43,579 GenProg runs. At least one run produced a patch that passed all of the training suite for 1,044 of the buggy versions, yielding a GenProg success rate of *generating a training-suite-passing patch* of 86.1%.

Not every GenProg run produced a patch. Particularly, since GenProg uses code elsewhere in the program to generate the patches, some versions simply didn't have enough material to work with. In these cases, GenProg timed out. In other cases, (due to an internal GenProg bug), GenProg claimed to have produced a patch that did actually pass all of the training suite. Overall, 50.0% (21,779 of the 43,579) of GenProg runs produced a patch.

By testing GenProg's effectiveness on six different programs, our experiments achieved a certain level of diversity in the types of programs GenProg attempted to fix. The programs include a variety of different program constructs, such as loops, conditions, character and numerical calculations. We also wanted to achieve a variety of levels of severity of the bugs. As described in Section 4.3, the buggy versions' training suites had varying numbers of failing tests, from just one to half of all the tests in the suite.

To understand what factors about the buggy version make it easier for GenProg to generate patches, we conducted a careful regression modeling of the bugginess and the complexity of the buggy version with GenProg's success rate of generating a patch. We tried several covariates: the size of the buggy version; the McCabe complexity of the buggy version; the training suite size; and the initial white-box passing rate of the buggy version. We built linear regression models allowing for each program to have a fixed effect (fixed effect modeling). While there were significant differences between the programs, none of the covariates had any significant effects on the GenProg's ability to patch the buggy programs.

As an example illustration of the lack of any systematic dependency, Figure 3 relates GenProg's success rate (y-axis) to the fraction of the white-box tests the buggy version passes (x-axis). Each data point represents all the GenProg runs on a single buggy version and its training suite. The shape of the data points represents the size of the training set (see legend). The data confirms visually that there is no clear relationship between the two plotted variables. The success rate of GenProg varies quite widely. The greater apparent spread in the y-axis values for some x-axis values is largely due to a larger number of samples in that region.

To answer Research Question 1, we conclude that GenProg is able to patch most (86%) of the buggy versions, but the bugginess (or indeed, the size, or McCabe complexity) of the program does not correlate with GenProg's success.

GenProg's general success in these experiments did produce adequate data to evaluate the next two research questions.

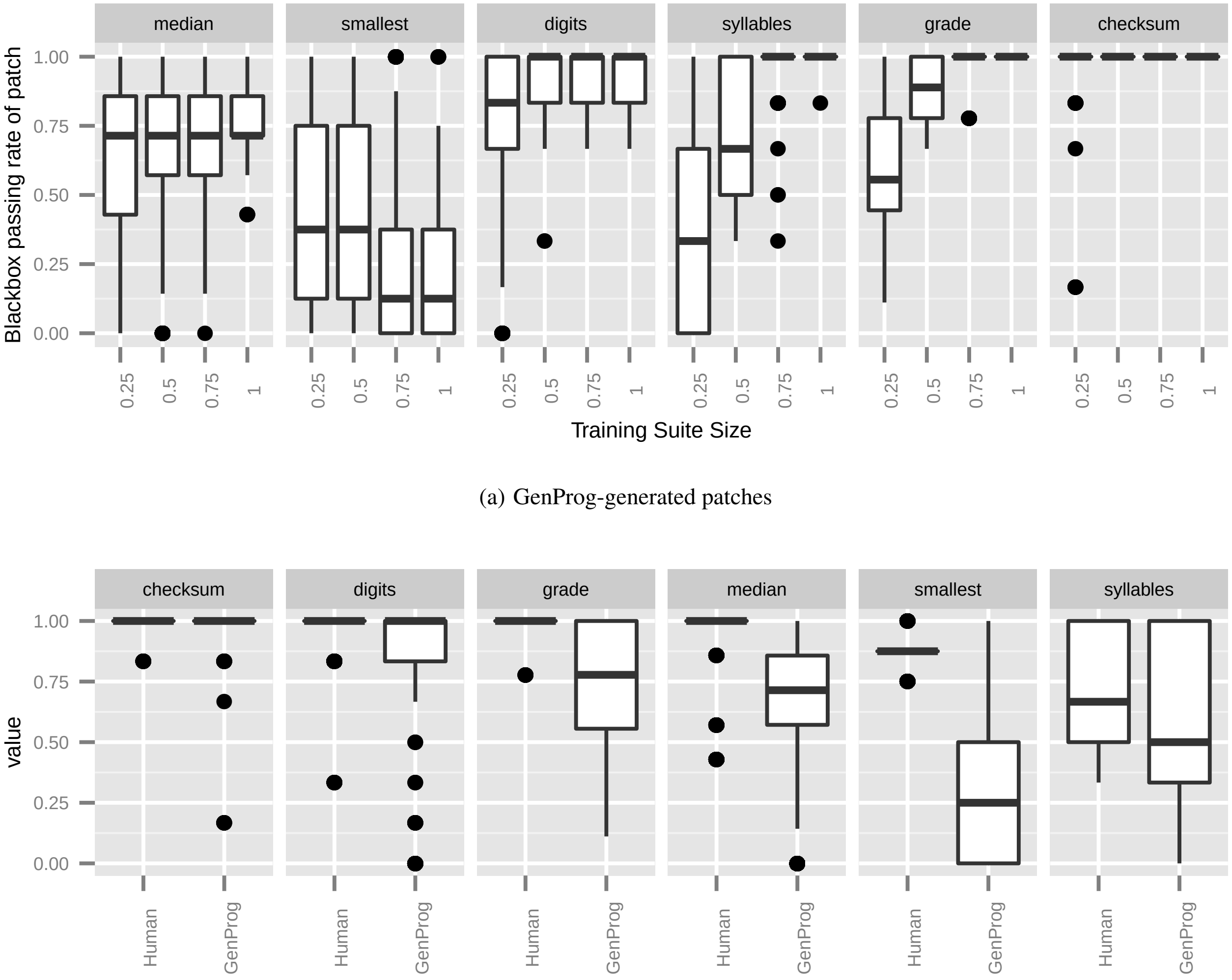
5.2 GenProg Solution Specificity

Using the large sample of successfully patched programs from Section 5.1, we next study the *specificity*, or *overfitting* of GenProg-generated patches. Specifically, we use the 21,779 patches produced by GenProg, for six different programs, under various choices of training suites, and evaluate their ability to generalize to the black-box suites designed to encode the program requirements.

Each of the 21,779 patched programs passes its own training suite. For each program, Figure 4 relates the size of the training suite to the fraction of black-box tests the patched version passes. The range of values shown within, e.g., the 0.25 box plot for *median* indicates distribution of rates at which the versions with 25%-sized training suites pass the black-box suites. Thus, at the 25% level, for *median*, the median is at about 0.74, indicating that half the programs trained on 25%-sized training suites pass 74% of the black-box tests.

For three programs, syllables, grade, and checksum, the larger training suites lead to patched programs that pass virtually all of the black-box tests, and thus do not overfit. For smaller training suite sizes, as expected, the patches fail more black-box tests because the training suite encodes less of the desired behavior. In regression modeling, for syllables and grade, the size of the training suite has a strong positive effect on the black-box pass rate. For the other four programs, the model's explanatory power was low; median and digits exhibited a positive effect, and smallest a negative one. Meanwhile checksum had no variance in the black-box pass rate.

4(b) compares the black-box passing rate of the human-written and GenProg-generated patches. The human-written patches are never worse, and for five of the six programs, dominate the GenProg-generated patches (statistically significant, corrected p< 0.01)



(a) GenProg-generated patches

(b) Human-written vs. GenProg-generated patches

Figure 4: (a) Black-box passing rate of *GenProg-patched* programs with a given training suite size. For checksum**,** grade**, and** syllables**, patched programs starting at higher levels pass virtually all black-box tests; for the rest, GenProg-generated patches based on subsets of the white-box suites are not as successful with black-box suites. (b) Human-written patches for the same buggy versions dominate the GenProg-generated counterparts, never overfitting more, and for five of the six programs, overfitting significantly less.**

To answer Research Question 2, we conclude that human-written patches overfit significantly less and dominate the GenProg-generated patches. GenProg quite often overfits to the training suite of tests: While for half the programs, GenProg patches that pass all of the white-box suite tests also pass all the black-box suite tests, for the other half, the solutions overfit to the white-box suites and fail many, sometimes most of the black-box suites.

To answer Research Question 3, we conclude that there is some positive effect of larger training suites on reducing overfitting: For two of the six programs, the effect is strong, whereas for the others, the effect is mixed and weak.

5.3 GenProg Solution Diversity

Many of the buggy versions gave rise to multiple GenProg patches. While individual patches tended to overfit to the training suites (recall Section 5.2), we wish to explore whether the diversity within these patches is sufficient to employ an n-version technique (recall Section 3.4) to produce solutions that do not overfit.

The n-version programs are plurality votes of more than three generated patches for each buggy version (again, recall Section 3.4).

Figure 5 compares the black-box passing rates of the buggy versions (in red), n-versions (in yellow), and human-written patches (in blue). As before, the results are split by program, and the x-axis indicates proportion of the white-box tests passed by the starting buggy student program, from which the n-patched versions arose.

The red data points represent the fitness of GenProg's starting points from which it attempts to generate a patch. Remarkably, in our experiments, the n-version solutions are *never* better than this starting point in a statistically significant way ($p < 0.01$). Further, the human-written patches outperform the n-version solution for

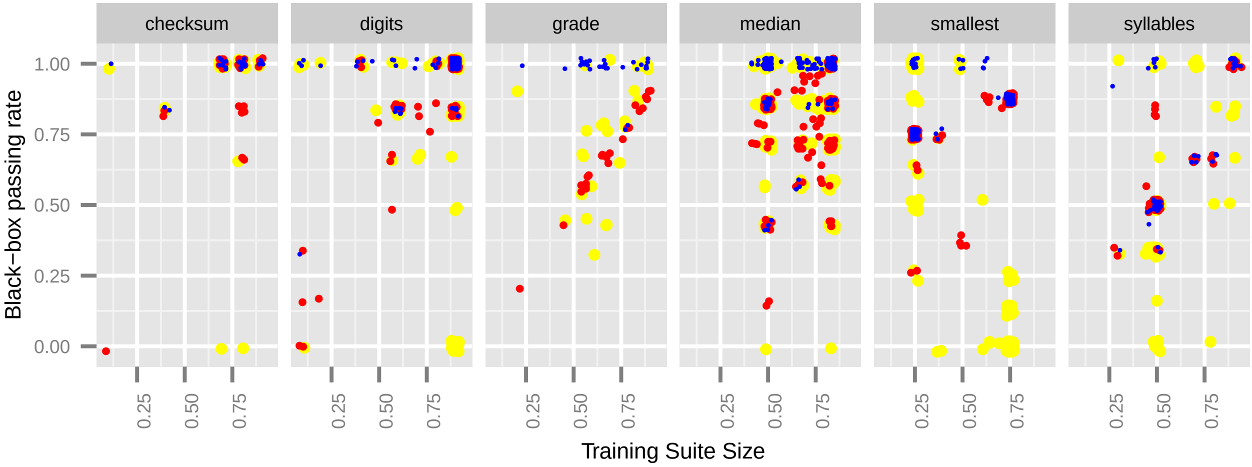


Figure 5: Comparison of the black-box suite passing rates of the buggy versions (red), the human-patched versions (blue), and the n-version programs made up of n GenProg-patched solutions (yellow). The data is jittered randomly to ameliorate overlaps. Statistical testing indicate that human-patched versions are generally the best and overfit the least. The n-version programs are never better than even the buggy versions.

five of the six programs ($p < 0.01$). The only exception is checksum, for which both humans and n-versions do very well. When we compared the n-version output to the GenProg patches, however, they generally performed better ($p < 0.01$)

To answer Research Question 4, we conclude that there is very little diversity from which n-version solutions benefit. In most cases, n-version solutions still overfit significantly more than human-written patches; however, they do, overall, improve even on the original buggy versions.

5.4 GenProg Patch Minimization Effectiveness

While GenProg solutions tended to overfit to the training suites, patch minimization *may* reduce overfitting, as discussed in Section 3.5. GenProg does not minimize its patches by default, but it does offer such an option. We reran GenProg on our set of buggy versions and training suites, just as in Section 5.1, but minimizing the patches. Figure 6 shows the black-box passing rates of the minimized GenProg-generated patches. These rates are nearly identical to the ones for unminimized patches (4(a)), so minimization has no significant effect on the rate.

To answer Research Question 5, we conclude that minimization has no significant effect on the quality of the patches, and thus does not reduce overfitting.

6. EXAMPLE: THE MEDIAN PROGRAM

This section describes one instance of a buggy student program and two patches that GenProg produced for it. We use these examples to highlight the ways that overfitting to a set of white-box test cases 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 7 shows the test suites used to evaluate the student submissions. The black-box tests were written by a human to encode the input space and holistic requirements of the assignment. The white-box tests were created using KLEE and a reference oracle implementation, as described in Section 4.2. The students had access to neither suite while completing the assignment; GenProg had access to only the white-box suite. We use the black-box suite to evaluate the adequacy of the final patches.

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 a correct solution, but it is convoluted and contains incorrect logic (e.g., the equality checks on lines 2 and 5). Despite this, it passes five of the six white-box and six of the seven the black-box tests. It fails on inputs with `n3` as the median, if the `n1` is greater than `n2` (such as the fifth black-box test, and the second white-box test in Figure 7). In these cases, the program fails to return an answer.

GenProg was able to generate several patches to this program, using the white-box suite for fitness evaluation. However, these patches vary considerably in quality. For example, consider the following low-quality, GenProg-generated patched program:

```
int med(int n1, int n2, int n3) {
  if ((n1==n2) || (n1==n3) || ((n3<n1) && (n1<n2)))
    return n1;
  if (n2<n1)
    return n3;
  if ((n2==n3) || ((n1<n2) && (n2<n3)) ||
            ((n3<n2) && (n2<n1)))
    return n2;
  if ((n1 < n3) && (n3 < n2))
    return n3;
}
```

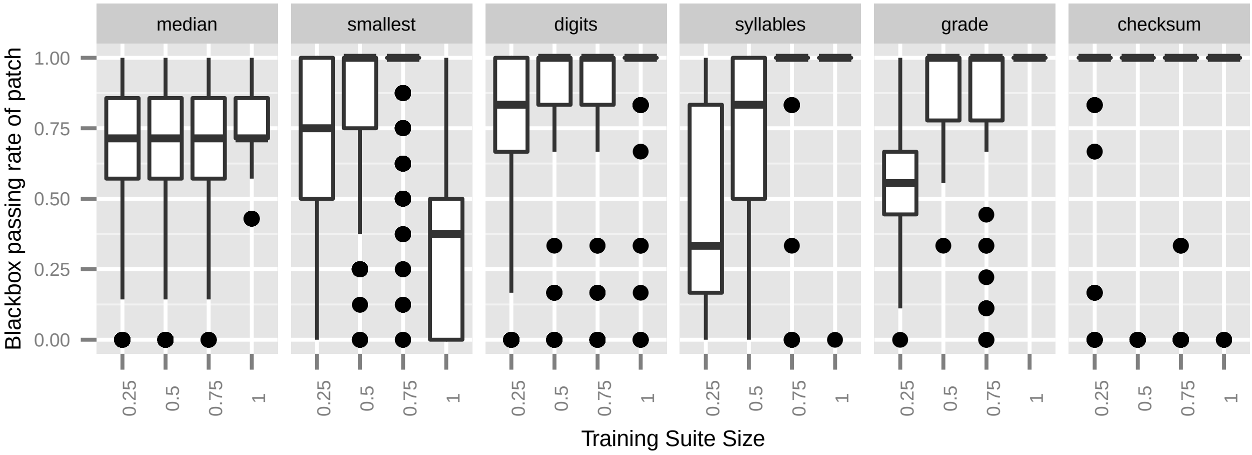


Figure 6: (a) Black-box passing rate of *minimized* GenProg-patched programs with a given training suite size. Comparing to 4(a), minimization does not have a significant effect on the passing rates.

One of the several conditions in the check on line 2 have been removed. As a result, this version of he program returns `n1` as the median only if it is coincidentally equal to either of the other two inputs, or if it is actually the median and `n3` is less than `n2`. If `n1` is not the median, but `n2` is less than `n1` (the check that has been moved to line 5), this code will (possibly incorrectly, possibly not, depending on the value of `n1`) return `n3` as the median.The patch does not affect the rest of the logic.

This patch addresses the original problem in the student's code, at least with respect to the white-box suite. This code returns the correct answer when: `n1` is the median and `n3` less than `n2`, `n2` is the median while being greater than `n1`, or `n3` is the median and `n2` is less than or equal to `n1`. Although this code passes all of the white box tests (improving on the original student submission), it actually passes fewer black-box tests, failing tests 3 and 6 in Figure 7.

This patch serves as an excellent example of overfitting to the fitness function. It highlights weaknesses in the white-box test suite: Many of the tests have repeated elements in the input, such as having multiple inputs be 0. 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. Consider the following patched program, also a result of running GenProg on the original student program with the white-box test suite.

```
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 has 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 to the problem, because by that point in the program logic, the only remaining option (captured by the `else`) *should* be that `n3` is a median.

Before submitting the final version, the student rewrote the logic of the code 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  {
3    if ((n2<=n1 && n1<=n3) || (n3<=n1 && n1<=n2))
4      return n1;
5    if ((n1<n2 && n2<=n3) || (n3<=n2 && n2<n1))
6      return n2;
7    if ((n1<n3 && n3<n2) || (n2<n3 && n3<n1))
8      return n3;
9  }
```

In this example, GenProg solutions overfit to the test suite, while the human-written patch did not. The fault lies in a large part with the test suite: The white-box test suite made available to GenProg did not encode some of the key behavior. However, partial test suites that underspecify the requirements are common in in the real world, so this situation is a quite realistic use of GenProg.

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 7: White- and black-box suites for the `median` subject program.

7. THREATS TO VALIDITY

Our experiments may not *generalize*. We only experiment with GenProg, one specific automatic program patcher. The results may not extend to other automatic program repair mechanisms. Second, all our subject programs were small student programs, with fairly small test suits. While these are experimentally interesting, in that they provide a very large set for conducting trials at scale, and for example, engaging in a large-scale test of the possibility of n-version programming, they are in fact a fairly small set. This small size may restrict the ability of GenProg to patch the programs; likewise the small test suite sizes may only explore the notion of fitness and training at a fairly small-scale and coarse levels. Finally, for programs that GenProg could not patch on five tries, we stopped after only those tries. (If GenProg showed promise by working at least once, we allowed it more tries.) Five is a small number by the standards of standard metahueristic search algorithms. It is possible that more attempts would have revealed more GenProg-generated solutions. Our results may over-approximate GenProg's success on some bugs, while under-approximating it across a selection of bugs. Further, we used the default GenProg settings for consistency with the related work; a full parameter sweep is outside the scope of this investigation.

Our experimental design may not be readily repeatable. For example, our intention was that the black-box white-box suites represented different perspectives on the fitness of the program. 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.

8. RELATED WORK

Automatically fixing bugs is a promising area of research because of both the high cost associated with bugs [10] and the high cost of fixing the bugs manually [3, 10, 37]. More than a dozen automated bug-fixing techniques have emerged. ClearView [33] detects invariants of normal operating behavior (e.g., `readIndex` $\leq$ `bufferSize`), and creates repairs that force data to conform to those invariants. PAR [24] improves search-based patching techniques with manually mined pattens of human repairs to find suitable patches. SemFix [31] constructs patches from symbolic-execution constraints. AutoFix-E [41] uses human-written code contracts to construct patches, which has the benefit of providing proofs of correctness, with the downside of additional burden on the human developer (to provide the contracts). Meanwhile even just mutating code fault-localization techniques deem faulty can repair bugs [39]. It is possible to evolve tests and repairs together [5], which could address some of the issues we have outlined with test suites used for repair. Focusing on a particular kind of bug can allow specialized fixes. For example, unsafe integer use in C programs can be fixed by applying template of faulty uses and code transformations [15]. For atomicity violations, AFix [23] uses static analysis to try to fix multiple bugs with a single fix. The ARMOR [12] replaces buggy library calls with a different library calls that achieve the same behavior; the same can be done for web applications [13]. GenProg, discussed and evaluated extensively in this paper, is a more general approach that uses GP techniques to evolve C programs toward ones that pass a set of supplied tests [44]. Our work does not create a new bug-fixing technique, but rather evaluates an existing technique, GenProg, attempting to compare the patches it produces to human-written patches, and to identify areas that need improvement.

We have already discussed previous evaluations of GenProg in Section 2.

GP algorithms tend to generate code bloat, extraneous code that does not contribute to the fitness of the solution [20]. This increases solution size and, in the case of GenProg, potentially effects the untested program behavior. Bloat in GP is well understood [36], and GenProg's use of minimization attempts to mitigate it.

Overfitting is a well-studied problem in machine learning [30]. In asking if patch minimization reduces overfitting (Sections 3.5 and 5.4), we were hopeful that code bloat and overfitting were related, since they both affect the behavior not covered by the training suite. Our results suggest that if there is no such relationship. To the best of our knowledge, there has been no significant prior consideration of the relationship of bloat and overfitting in the domain of patch generation.

Search-based software engineering [22] uses search-based methods, such as evolution for software engineering tasks, such as developing test suites [29, 40], finding safety violations [2], refactoring [35], and project management and effort estimation [6]. Good fitness functions are critical to search-based software engineering, as is also the case with GenProg. Our findings indicate that using test cases as the fitness function leads to patches that overfit to that function, and do not generalize well to satisfy the requirements.

In our evaluation, we used an n-version approach [14] of combining multiple programs trying to solve the same problem. This approach has been shown to work poorly with human-written systems because the errors humans make when writing programs do not appear to be independent [4]. With automated bug fixing, this approach did improve slightly over using the individual patches, but still failed to produce desired behavior.

9. CONTRIBUTIONS

Automated bug-fixing techniques have the potential to significantly reduce costs and improve software quality and recent research has shown great promise. In this paper, we have evaluated GenProg, a well-known automated bug-fixing technique on a large set of student-written programs with naturally occurring bugs, and human-written patches. Our dataset consists of 1,212 bugs that occurred naturally during development of six programs, and our experiments generated 21,779 automated patches (using different test suites to drive the patching process). Our experiments conclude that GenProg was able to produce a patch for most (86%) of the bugs, but that these patches overfit to the test suite used in patching, and often failed to pass other tests that encode program requirements. Meanwhile, human-written patches *significantly* outperformed GenProg-generated patches on these other tests. Further, we found that providing GenProg with larger test suites sometimes decreased overfitting, but that the GenProg-generated patches lacked sufficient diversity to use a kind of n-version bagging, in which a plurality vote over a population of GenProg-generated patches outperforms a randomly chosen individual patch. Finally, we found that patch minimization did not affect overfitting. In the end, while automated bug-fixing is promising, much work remains in improving the quality of the generated patches before they equal that of human developers.

10. REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2007.

[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] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, Shanghai, China, 2006.

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

[7] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *International Conference on Software Engineering*, 2013.

[8] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-Constrained Models. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2011.

[9] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, June 1972.

[10] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 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] 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] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[17] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

[18] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Conference on Genetic and Evolutionary Computation*, pages 947–954, Montreal, Québec, Canada, 2009.

[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] 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.

[21] A.-R. Han and D.-H. Bae. Dynamic profiling-based approach to identifying cost-effective refactorings. *Information and Software Technology*, 55(6):966 – 985, 2013.

[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] 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.

[25] 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.

[26] 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.

[27] 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.

[28] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *International Conference on Software Engineering*, 2008.

[29] 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.

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

[31] 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.

[32] G. Paris, D. Robilliard, and C. Fonlupt. Exploring overfitting in genetic programming. In *Artificial Evolution*, volume 2936 of *Lecture Notes in Computer Science*, pages 267–277. Springer, 2004.

[33] 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.

[34] S. P. Reiss and M. Renieris. Encoding program executions. In *International Conference on Software Engineering*, 2001.

[35] 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.

[36] 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.

[37] Symantec. Symantec Internet security threat report. Trends for january 06–june 06. `http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf`, September 2006.

[38] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[39] V. D. W. Eric and 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.

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

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