Quality of Automated Program Repair on Real-World Defects

Manish Motwani, Mauricio Soto, Yuriy Brun, *Senior Member, IEEE*, René Just, and Claire Le Goues, *Member, IEEE*

Abstract—Automated program repair is a promising approach to reducing the costs of manual debugging and increasing software quality. However, recent studies have shown that automated program repair techniques can be prone to producing patches of low quality, overfitting to the set of tests provided to the repair technique, and failing to generalize to the intended specification. This paper rigorously explores this phenomenon on real-world Java programs, analyzing the effectiveness of four well-known repair techniques, GenProg, Par, SimFix, and TrpAutoRepair, on defects made by the projects' developers during their regular development process. We find that: (1) When applied to real-world Java code, automated program repair techniques produce patches for between 10.6 and 19.0 percent of the defects, which is less frequent than when applied to C code. (2) The produced patches often overfit to the provided test suite, with only between 13.8 and 46.1 percent of the patches passing an independent set of tests. (3) Test suite size has an extremely small but significant effect on the quality of the patches, with larger test suites producing higher-quality patches, though, surprisingly, higher-coverage test suites correlate with lower-quality patches. (4) The number of tests that a buggy program fails has a small but statistically significant positive effect on the quality of the produced patches. (5) Test suite provenance, whether the test suite is written by a human or automatically generated, has a significant effect on the quality of the patches, with developer-written tests typically producing higher-quality patches. And (6) the patches exhibit insufficient diversity to improve quality through some method of combining multiple patches. We develop JaRFly, an open-source framework for implementing techniques for automatic search-based improvement of Java programs. Our study uses JaRFly to faithfully reimplement GenProg and TrpAutoRepair to work on Java code, and makes the first public release of an implementation of Par. Unlike prior work, our study carefully controls for confounding factors and produces a methodology, as well as a dataset of automatically-generated test suites, for objectively evaluating the quality of Java repair techniques on real-world defects.

Index Terms—Automated program repair, patch quality, objective quality measure, Java, GenProg, Par, TrpAutoRepair, Defects4J

1 Introduction

Automated program repair holds the potential to improve software quality while simultaneously reducing the reliance on costly manual effort. For example, Facebook uses two automated program repair tools, SapFix and Getafix, on their production code to suggest defect patches [9], [89]. However, recent work examining the quality of automated program repair has found that patches produced by many automated program repair techniques are often of low quality [122] and not semantically equivalent to developer-written patches [114]. In particular, our earlier work [122] found that patches produced by GenProg [77], TrpAutoRepair [111], and AE [132] typically pass only 68.7, 72.1, and 64.2 percent of independent tests not used to create the patch, respectively. This both raises an important concern about the practical usability of modern automated repair techniques, and drives research toward building techniques that produce higher-quality patches [68], [86], [88], [94].

Automated program repair techniques typically start with a program version and a set of passing and failing tests, and then modify the program version until finding a set of modifications (a patch) that makes all the tests pass. The underlying issue is that the set of tests provides a partial specification of the desired behavior, and thus the produced patches may overfit to those tests. For example, while, typically, many patches in a technique's search space pass the supplied tests, relatively few are equivalent to the developer-written patch [88], [114]; the automated repair technique has no way of knowing which is the better patch to return.

Our prior work introduced an objective methodology for evaluating the quality of a patch and had successfully applied it to a set of very small programs written by novice developers in an introductory programming course [122]. While that work identified important shortcomings of automated program repair techniques, its results may not generalize beyond the very small and simple programs. That study only considered two generate-and-validate (*G&V*) repair techniques, did not control for confounding factors, and used test suite size as a proxy for coverage. By contrast, this work performs a detailed study with four *G&V* repair techniques on real-world defects in real-world, large, complex projects employing

- *Manish Motwani and Yuriy Brun are with the College of Information and Computer Sciences, University of Massachusetts Amherst, Amherst, MA 01003-9264. E-mail: {mmotwani, brun}@cs.umass.edu.*
- *Mauricio Soto and Claire Le Goues are with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: {msotogon, clegoues}@cs.cmu.edu.*
- *Rene Just is with the Paul G. Allen School of Computer Science and Engineering, University of Washington, Seattle, WA 98195-2350. E-mail: rjust@cs.washington.edu.*

Manuscript received 21 Apr. 2019; revised 10 Mar. 2020; accepted 18 May 2020. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Manish Motwani.*)*
Recommended for acceptance by E. Bodden.
Digital Object Identifier no. 10.1109/TSE.2020.2998785

rigorous statistical analyses, properly measuring coverage, and controlling for confounding factors. We use 5 programs with 357 defects created during real-world development from the Defects4J benchmark [66]. We selected four representative repair techniques and a diverse benchmark of defects to increase the likelihood that our results generalize. We answer six research questions:

RQ1 Do *G&V* techniques produce patches for real-world Java defects?

Answer: Yes, although less often than for C defects.

RQ2 How often and how much do the patches produced by *G&V* techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification?

Answer: Often. For the four techniques we evaluated, only between 13.8 and 41.6 percent of the patches pass 100 percent of an independent test suite. Patches typically break more functionality than they repair.

RQ3 How do the coverage and size of the test suite used to produce the patch affect patch quality?

Answer: Larger test suites produce slightly higher-quality patches, though, surprisingly, the effect is extremely small. Also surprisingly, higher-coverage test suites correlate with lower quality, but, again, the effect size is extremely small.

RQ4 How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit?

Answer: The number of failing tests correlates with slightly higher quality patches.

RQ5 How does the test suite provenance (whether it is written by developers or generated automatically) influence patch quality?

Answer: Test suite provenance has a significant effect on repair quality, although the effect may differ for different techniques. In most cases, human-written tests lead to higher-quality patches.

RQ6 Can overfitting be mitigated by exploiting randomness in the repair process? Do different random seeds overfit in different ways?

Answer: The patches exhibit insufficient diversity to improve quality through some method of combining multiple patches.

Our methodology for measuring patch quality relies on an independent test suite that is not given to the repair technique to produce a patch. The independent test suite captures (again, partially) some of the specifications not captured by the original test suite given to the repair technique, and thus its passing rate independently evaluates the quality of the patch. The alternative to this methodology is a manual inspection of the patch, (e.g., [114]), but two independent recent studies [72], [140] have empirically demonstrated that our independent-test-suite-based methodology is more reliable and more objective than manual inspection.

Prior studies of quality of automated program repair have either used manual inspection for quality assessment [107], [122], [131], or have focused on small programs and relatively-easy-to-fix defects [122], [140]. Some studies did use a 224-defect subset of the same benchmark of real-world programs we use, but used manual inspection for quality assessment and, unlike our work, assessed tools' ability to produce patches and efficiency of patch production, but did not identify the factors that affect patch quality (RQs 3–6) [42], [90].


http://JaRFly.cs.umass.edu/

Our work overcomes two considerable new engineering challenges. First, employing the objective, independent-test-suite-based evaluation of patch quality, requires the creation of high-quality, automatically-generated test suites for real-world Java projects. We develop a methodology for using today's state-of-the-art test-suite generation techniques and overcoming their shortcomings to produce high-quality suites, and we release both the methodology and the generated test suites. Second, many automated program repair techniques are designed and implemented for C (e.g., GenProg and TrpAutoRepair) and Par [69], designed and implemented for Java, was never released. We build JaRFly, the Java Repair Framework, which simplifies the implementation of Java techniques for genetic improvement (including but not limited to genetic improvement techniques for program repair), and release Java-based implementations of GenProg, Par, and TrpAutoRepair. Our implementations of GenProg and TrpAutoRepair are the first that faithfully follow the original techniques' designs, improving prior attempts at replicating these techniques for Java. Our release of the Par implementation is the first ever public release of Par. JaRFly is the first framework of its kind that can handle the entire Defects4J dataset, including the Closure compiler subject program.

The main contributions of our work are:

- An empirical evaluation of quality of program repair on real-world Java defects, which outlines shortcomings and establishes a methodology and dataset for evaluating quality of new repair techniques' patches on real-world defects to promote research on high-quality repair.
- A methodology for evaluating patch quality that fixes numerous shortcomings in prior work, properly controlling for potential confounding factors.
- A dataset of independent evaluation test suites for Defects4J defects, and a methodology for generating such test suites. Augmenting existing Defects4J defects with two, independently created test suites can aid not only program repair, but other test-based technology.
- Java Repair Framework (JaRFly), a publicly released, open-source framework for building Java *G&V* repair techniques, including our reimplementations of GenProg, Par, and TrpAutoRepair. JaRFly is designed to allow for easy combinations and modifications to existing techniques, and to simplify experimental design for automated program repair on Java programs. `http://JaRFly.cs.umass.edu/`

The rest of this paper is structured as follows. Section 2 describes the background of automated program repair. Section 3 introduces JaRFly. Section 4 details the dataset of real-world defects used in our study and our methodology for creating high-quality test suites. Section 5 empirically evaluates four automated program repair techniques with respect to the quality of the patches they produce on real-world defects. Section 6 discusses the implications of our results, suggests future directions for research, and describes the limitations of our choices of subject repair tools and defects. Finally, Section 7 places our work in the context of related research, and Section 8 summarizes our contributions.

2 Automated Program Repair

Automated program repair techniques' goal is to convert an existing program that nearly satisfies a specification into one that fully satisfies it. This can be done for many types of specifications, e.g., contracts [107], [131], a reference implementation [92], or, by far most commonly, tests. This paper focuses on test-based program repair.

Unfortunately, tests provide only a partial specification of the desired behavior, and, as such, producing a patch that passes all the tests might break other untested or under-tested functionality. Patches that pass all supplied tests but do not generalize to the intended specification are said to be of low quality and to overfit to the test suite used to produce them. Section 2.1 will provide background on automated program repair, and Section 2.2 will explain methods for evaluating patch quality.

2.1 *G&V* and Synthesis-Based Repair

Automatic program repair techniques can be classified broadly into two classes: (1) *Generate-and-validate* (*G&V*) techniques create candidate patches (often via search-based software engineering [57]) and then validate them, typically through testing (e.g., [5], [29], [36], [39], [68], [69], [83], [93], [101], [114], [120], [125], [132], [133]. (2) *Synthesis-based* techniques use constraints to build patches via formal verification, inferred or programmer-provided contracts, or specifications (e.g., [64], [107], [131]). *Runtime program repair* techniques (e.g., [23], [24], [37], [38], [108] self-heal the execution at runtime and typically do not produce source-code patches, and are orthogonal to the above classification. This paper focuses on *G&V* techniques, and neither synthesis-based nor runtime-repair techniques. Prior work has considered overfitting in synthesis-based repair techniques [76], albeit only on small programs. While both synthesis-based and *G&V* techniques share high-level goals, they work best in different settings, and have different limitations and challenges.

Test-driven *G&V* techniques are a particularly interesting subject of exploration, as they (e.g., Clearview [108], GenProg, Par, and Debroy and Wong [36]) have been shown to repair defects in large, real-world legacy software. Meanwhile, formal specifications and contracts are relatively rare in practice. Although new projects appear to be increasingly adopting contracts [46], their penetration into existing systems and languages remains limited. Few maintained contract implementations exist for widely-used languages such as C. For example, 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 are often idiosyncratic to the point of not adhering to the specifications of their host language [15], it might not be possible even to add contracts to such projects.

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

We chose four representative *G&V* repair techniques for our analysis. There are many existing *G&V* repair techniques, often with similar performance. However, an underlying theory of *G&V* repair suggests that analysis of a set of these techniques should generalize to others [132]. Section 6 discusses the generalizability of our results.

GenProg [77], [133] uses a genetic programming heuristic [71] to search the space of candidate repairs. Given a buggy program and a set of tests, GenProg generates a population of random patches by using statistical fault localization to identify which program elements to change (those that execute only on failing test cases or on both failing and passing text cases), and selecting elements from elsewhere in the program to use as candidate patch code. The fitness of each patch is computed by applying it to the input program and running the result on the input test cases; a weighted sum of the count of passed tests informs a random selection of a subset of the population to propagate into the next iteration. These patch candidates are recombined and mutated to form new candidates until either a candidate causes the input program to pass all tests, or a preset time or resource limit is reached. Because genetic programming is a random search technique, GenProg is typically run multiple times on different random seeds to repair a bug.

Par [69] performs search by applying 12 fix templates — automatic program editing scripts created based on the fix patterns identified from developer fixes — in the locations they can be applied that are also identified as likely faulty by statistical fault localization.

SimFix [63], a more recent technique, mines code patterns (similar to Par templates) from frequently occurring code changes from developer-written patches. Then, in the project with the defect SimFix is attempting to repair, SimFix identifies code snippets that are similar to the code SimFix has localized the defect to. SimFix defines similarity using structural properties, variable names, and method names. SimFix ranks the code snippets by the number of times the mined patterns have to be applied to the snippet to replace the buggy code. SimFix then selects the snippets (one at a time) from the ranked list of top 100 snippets, applies the pattern-based modifications to produce a candidate patch, and validates the patch against tests created using a test purification technique [139]. While the original paper describes SimFix stopping once a patch that passes the test suite is found [63], the implementation [62] generates multiple patches that pass at least one of the purified originally-failing tests. In this paper, we use all the found patches for our analyses.

TrpAutoRepair [111] uses random search instead of genetic programming to traverse the search space of candidate solutions. Instead of running an entire test suite for every patch, TrpAutoRepair uses heuristics to select the most informative test cases first, and stops running the suite once a test fails. TrpAutoRepair limits its patches to a single edit. It is more efficient than GenProg in terms of time and test case evaluations [111]. The same approach is also called RSRepair [112], and we refer to the original algorithm name in this paper.

There are four key challenges that *G&V* must overcome to find patches [132]. 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 that any one of them is changed at a given time describes the *fault space* of a particular program repair problem. GenProg, Par, SimFix, and TrpAutoRepair tackle this challenge by using existing fault localization techniques to identify good repair candidates. SimFix increases the accuracy of GZoltar [22], an existing fault localization technique, by using a test purification technique [139] that removes assertions unrelated to the bug from the failing tests, as well as source code statements related to those unrelated assertions. 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 TrpAutoRepair tackle this challenge using the observation that programs are often repetitive [10], [51] and logic implemented with a bug in one place is likely to be implemented correctly elsewhere in the same program. GenProg and TrpAutoRepair therefore limit the code changes to deleting and copying constructs from elsewhere in the same program. Par instantiates a set of repair *templates* constructed based on a manual inspection of a large set of developer edits to open source projects. SimFix similarly uses templates mined from developer-written patches, also limiting code changes to snippets from the same program which are similar structurally, or through variable or method names, to the code being replaced. Third, there are many ways to edit the code snippets identified by the *fix space* so as to patch the bug. These edits, called mutation operators, define the *repair strategy*. GenProg and TrpAutoRepair use three mutation operators, selected uniformly at random, *append* candidate snippet, *replace* the buggy region with the candidate snippet, and *delete* the buggy region. GenProg also allows for a *crossover* operator that combines parts of two candidate snippets. Par uses 12 mutation operators, chosen uniformly at random, each one corresponding to its 12 fix templates. SimFix uses the code patterns mined from the existing developer-written patches and selects the candidate snippets that requires fewer modifications using the mined code patterns. Fourth, selecting the tests to be executed to evaluate a candidate patch defines a repair technique's *test strategy*. GenProg and Par sample 10 percent of the tests using random sampling for internal computations, and only the full test suite for promising candidates. TrpAutoRepair uses heuristics to select the most informative test cases first, and stops running the suite once a test fails. SimFix executes all the failing tests first and, only if all those pass, continues to execute the passing tests.

GenProg, Par, SimFix, and TrpAutoRepair share sufficient common features to allow consistent empirical and theoretical comparisons. This allows us to focus on particular experimental concerns and mitigates the threat that unrelated differences between the algorithms confound the results.

2.2 Evaluating Repair Quality

In 2013, Brun *et al.* [20] demonstrated that automated program repair is prone to producing patches that overfit to the test suites it has access to. Within the space of possible program modifications, many programs (and, thus, patches) exist that pass all the supplied tests. While some of these programs encode the desired behavior for all possible inputs, many fail to encode desired behavior on at least some inputs not represented by the tests. Those other programs fail to generalize to the unwritten, intended specification and result in low-quality patches. This phenomenon of automated program repair producing patches that satisfy the partial specification of the supplied test suite, but failing to generalize is called overfitting [122].

Since then, research has measured the degree to which *G&V* patches overfit and what factors affect that overfitting on small C programs [122], how often *G&V* patches disagree with developer-written patches [114], how often overfitting happens in Java repair [42], [90], the space of possible patches and the concentration of correct ones [87], and so on. Further, research has attempted to improve on the quality of the patches produced by using semantic search to increase the granularity of repair [68], condition synthesis [86], learning patch generation patterns from human-written code [88], and automated test case generation [135]. Other research has found that overfitting is not unique to *G&V* C repair, with synthesis-based repair also overfitting to the supplied partial specification [76]. Even when repair uses manually-written contracts as the desired behavior specification, which are more complete than tests, it still overfits, producing correct patches for only 59 percent of the defects [107].

There are two established methods for evaluating quality of program repair, using an independent test suite not used during the construction of the repair [20], [122], and manual inspection [90], [114], typically for equivalence with a developer-written patch (though manual inspection has been used to measure how maintainable the patches are [50] and how likely developers are to accept them [69]). The two methodologies are complementary. Intuitively, the methodology that uses an independent test suite is more objective, whereas manual inspection is more subjective and can be subject to subconscious bias, especially if the inspectors are authors of one of the techniques being evaluated. A recent study found that manual-inspection-based quality evaluation can be imprecise [72], while independent-test-suite-based quality evaluation is inherently partial, as the independent test is a partial specification. As a result, manual evaluation of quality can imprecisely label patches as correct and incorrect. The test-suite-based evaluation cannot be imprecise, but may be incomplete, potentially mislabeling some patches as correct but never labeling a correct patch as incorrect.

In this paper, we select to use the test-suite-based quality evaluation method because (1) it is objective and reproducible in a fully automated manner, (2) can scale to complex, real-world defects in real-world systems, which are the focus of our work (whereas manual inspection would require using the projects' developers with intricate project knowledge). Since this methodology necessarily underestimates overfitting (it never labels a correct patch as incorrect) [72], our findings of overfitting are, at worst, conservative.

3 JaRFly: The Java Repair Framework

This section describes JaRFly, our open-source framework for implementing techniques for automatic search-based improvement (or *genetic improvement*) of Java programs. Genetic improvement approaches reuse existing software as input to metaheuristic search. The search goal is to identify variants of that input software that improve on the software according to some criterion (e.g., functionality, performance) [109]. JaRFly is publicly available at `http://JaRFly.cs.umass.edu/` to facilitate researchers and practitioners building search-based improvement approaches for Java programs. The implementation includes reimplementations of GenProg [77] and TrpAutoRepair [111] for Java (original releases of these tools were for C programs), and releases the first public reimplementation of Par [69].

JaRFly's novelty and utility lie in the way it decouples the fundamental components of metaheuristic search and allows developers to specify just those fundamental components, taking care of the rest of the approach implementation. These components are problem representation, fitness function, mutation operators, and search strategy [58]. JaRFly provides high-level extension points for each of these fundamental components, which differentiates it from prior frameworks that support implementing Java-based repair techniques [91].

JaRFly simplifies the process of implementing genetic improvement approaches for Java programs. JaRFly handles parsing Java programs into a specified representation, and metaheuristic search over variants within that representation using specified mutation operators, search strategy, and fitness function. JaRFly allows the user to specify these representations, mutation operators, search strategies, and fitness functions by selecting from a set of already implemented options, or by extending with new versions via explicit extension points.

JaRFly improves on prior frameworks that support implementing Java-based repair techniques [91] by making these fundamental components explicit and supporting their extensions explicitly, while also handling a wider range of Java programs. For example, JaRFly can operate over the Closure compiler subject program from the Defects4J dataset, whereas prior frameworks cannot [91]. We next detail JaRFly's four fundamental components of metaheuristic search.

Problem Representation. The first and perhaps most fundamental design choice in applying metaheuristic search to a software engineering problem is deciding how to represent the problem such that it is amenable to symbolic manipulation. The most common representation choice in genetic improvement applications is the *patch representation*, in which an individual candidate solution is represented as a variable-length sequence of edits to the original program [77], [78]. In addition to Java, variations of and improvements on this representation choice can target Python [2] and C [103], [104] programs. Prior to the development of the patch representation, genetic-programming-based program repair operated over problems represented as a fixed-length weighted path through the program represented as an abstract syntax tree [48], [133]; as is typical in metaheuristic search, representation choice influences search success and efficiency [78]. By making this representation an explicit choice, and extension point, JaRFly enables developers to both pay proper attention to the choice of representation and to evaluate multiple representation choices.

JaRFly's `Representation` interface exports functionality for manipulating and evaluating a candidate solution in the context of a search-based program improvement approach. This includes support for

1) querying variant-specific localization information,
2) evaluating fitness, such as serializing a variant to disk and compiling it, or running one or more test cases against a given variant, tasks common to most genetic improvement approaches, depending on fitness function, and
3) assessing the validity of and applying mutation operators to the particular variant.

To that end, JaRFly's `Representation` is parameterized by a mutation interface that provides functionality for editing arbitrary Java programs.

JaRFly provides prebuilt implementations of (1) an abstract superclass that supports caching and serialization of common representation-independent intermediate data, such as a fitness cache, and (2) a classic patch representation for program repair problems in Java. The currently-implemented patch representation is a variable-length list of indivisible mutation operators, such as "Insert statement S at location L"; mutating this representation adds a new edit to the end of the current variant. It is straightforward to implement other choices without requiring major refactoring of the framework. For example, Oliveira *et al.* [103], [104] propose a novel patch-based representation that decouples the fault, operator, and fix spaces, with implications for crossover (but no other components of the search strategy); this could be achieved for Java in our framework by specializing the present patch-based representation (specifically the `getGenome` method) and implementing the new crossover operators in dedicated methods in the `Population` module.

Fitness Function. Applying metaheuristic search to a software engineering problem requires a *fitness function* to determine the fitness of a variant. Thus, this function must operate on the representation. JaRFly makes the choice of the fitness function explicit.

The most typical fitness function in modern repair approaches is a weighted sum of the number of test cases passed by a program variant. Sampling can reduce the computational cost of this fitness function [47]. Alternative fitness functions for program repair typically combine test cases with another objective, such as in a multi-objective search strategy. These alternative objectives can include a variant's similarity to patches in a dataset of previous developer-written patches [75], or its intermediate semantic distance according to a set of learned invariants over intermediate program state [40], [47] or according to memory values [34] from either the original program or the rest of the population.

JaRFly provides an extensible, representation-agnostic `Fitness` module that, by default, implements and provides configuration options for multiple common fitness strategies from the genetic improvement literature. These strategies include test execution at different levels of JUnit granularity (individual JUnit method, or entire JUnit class), and configuration options for test sampling (including generational versus individual sampling, and a configurable sample rate), and test selection (sampled, heuristically modeled [111], [132], or test to first failure). JaRFly's `Fitness` interface is agnostic to the underlying testing methodology, so it is not limited to using JUnit for fitness calculation. `Fitness` provides, by default, the idea of a (potentially dynamically-updated) *test model*, supporting experiments and extensions focused on more intelligent test selection and prioritization. JaRFly, moreover, extends (in a non-default branch) `Fitness` to evaluate and provide additional values, such as an experimental diversity-based metric [40], in the context of a multi-objective search strategy (NSGA-II [35]) extended from the `Search` module. Other measures of fitness, such as via comparison to a historical dataset of patches [75], can similarly extend `Fitness.testFitness` for more specialized, non-test-driven metrics.

Mutation Operators. Metaheuristic search requires a set of manipulation operators applicable to the selected representation. JaRFly provides the `EditOperation`abstraction, parameterized by a rewriter engine that can modify arbitrary Java programs. JaRFly's default implementation uses the Eclipse JDT API to perform rewriting. An `EditOperation` is instantiated at a particular (abstract) `Location`, and may contain one or more abstract `Hole`s that need to be filled in with suitable code. For example, an `Append`operation can be instantiated at any statement in a Java location; it has a single `Hole` that must be filled in by a piece of code that may be appended there.

JaRFly implements all statement-level edit operations used by GenProg and TrpAutoRepair and all Par fix templates, including the optional ones from `https://sites.google.com/site/autofixhkust/home/`, not included in the original paper [69]. Both GenProg and TrpAutoRepair construct modifications by reusing code from elsewhere in the program under repair. The `Representation` enforces this type of modification, providing information on legal `Location`s and code bank code that can be used to fill in `Hole`s for a particular variant. Meanwhile Par uses 12 fix templates — automatic program editing scripts created based on the fix patterns identified from developer-written patches. As with the coarser-grained operations used by GenProg and TrpAutoRepair, the `Representation` provides the possible values to fill in `Hole`s in Par's fix templates, such as which variable should be checked for `null` in the null-check-insertion template.

Some `EditOperation`s cannot be applied at all `Location`s. For example, an `Append` operation cannot insert code that references out-of-scope variables, or the result will not compile. JaRFly creates `EditOperation`s via a helper `JavaEditFactory`, which queries a variant via its `Representation` interface for information to determine the edit's legality. JaRFly implements a set of static semantic checks that can identify edits that will be rejected by the compiler. Previous work demonstrated that static semantic checks improve efficiency in genetic programming repair for C programs [78]. Java's compiler is substantially stricter than most C compilers, requiring commensurately more complex static checks to avoid invalid mutations.

Although we use the released SimFix implementation for our experiments, the mutation operators considered by SimFix could be implemented further as abstractions or extensions of this paradigm. Mutation operators are typically associated with weights that inform their selection and application. In the default implemented algorithms, these weights are fixed throughout the search strategy. However, they are customizable by design, such as via a machine-learned model of edit frequency drawn from historical, developer-written patches [88], [123].

Search Strategy. The choices of representation and mutation operators represent the space of possible variants metaheuristic search can explore, and the choice of fitness function represents the objective shape of that search space. The *search strategy* defines the path through the space the metaheuristic search uses to optimize the objective.

Common search strategies include local search, random search, and genetic programming. JaRFly's `Search` interface provides a representation-agnostic extension point for implementing search strategies, and implements five strategies, to facilitate comparison and customization. The implemented strategies are a random search, a weighted brute force single-edit search, an oracle search, a genetic programming heuristic, and NGSA-II [35], a multi-objective evolutionary search strategy.

In addition to these four fundamental components of the metaheuristic search, JaRFly includes implementation and support for other common and important interfaces and utilities for search-based program modification:

Population Manipulation. JaRFly implements crossover and selection strategies common in source-level evolutionary program manipulation. The implemented crossover strategies include one-point crossover, uniform crossover [133], and crossback crossover (crossover with the original unmodified representation) [133]. The one implemented selection strategy is tournament selection with configurable tournament sizes. JaRFly contains extension points to make adding new crossover and selection operators straightforward and independent of representation. Additionally, JaRFly allows setting the proportional mutation rate as a top-level configuration option.

Localization and Code Bank Management. Fault and fix localization are common concerns in search-based program repair or improvement. JaRFly implements common weighted path localization with configurable path weights, facilities for reading in arbitrary localization data from a file, and an abstract class for implementing alternative localization strategies [113]. JaRFly uses the JaCoCo coverage library to compute coverage for the purposes of fault localization [44].

These facilities support significant (but straightforward) customization and investigation of all elements of a meta-

identifier	project	description	KLoC	defects	tests	test KLoC
Chart	JFreeChart	Framework to create charts	85	26	222	42
Closure	Closure Compiler	JavaScript compiler	85	133	3,353	75
Lang	Apache Commons Lang	Extensions to the Java Lang API	19	65	173	31
Math	Apache Commons Math	Library of mathematical utilities	84	106	212	50
Time	Joda-Time	Date- and time-processing library	29	27	2,599	50
total			302	357	6,559	248

Fig. 1. The 357 defect dataset created from five real-world projects in the Defects4J version 1.1.0 benchmark. We used `SLOCCount` to measure the lines of code (KLoC) counts (https://www.dwheeler.com/sloccount/). The *tests* and *test KLoC* columns refer to the developer-written tests.

heuristic search technique for program transformation. Implementing different metaheuristic search strategies (regardless of the search goal) requires specialization of a single `Search` class; investigating or isolating the effect of particular search features (such as selection, crossover or mutation rate, or the numerous other parameters influencing the traversal strategy in a genetic algorithm) requires the specialization of single methods, or the modification of existing top-level configuration options. These choices enable significant ongoing experimentation and specialization of the *search* component of a search-based or genetic improvement program modification strategy, without requiring reimplementation or modification of how programs under modification are represented, manipulated, or evaluated.

4 Real-World Defects and Test Suites

Our study requires real-world defects in real-world projects. Further, our study requires that each of these projects have not one but two high-quality test suites. Section 4.1 describes the Defects4J [66] dataset we use in our study, and Section 4.2 describes the methodology we followed to create test suites.

A replication package, with all data, code, and instructions necessary to replicate our results is available at `http://github.com/LASER-UMASS/JavaRepair-replication-package/`.

4.1 Real-World Defects

We used Defects4J version 1.1.0, which consists of 357 defects made by developers during the development of five real-world open-source Java projects. Fig. 1 describes the Defects4J defects and the projects they come from. Each defect comes with (1) one defective and one developer-repaired version of the project code; (2) a set of developer-written tests, all of which pass on the developer-repaired version and at least one of which evidences the defect by failing on the defective version; and (3) the infrastructure to generate tests using modern automated test generation tools. Each defective version is a real-world version of the code. This version, submitted to the project's version-control repository by the developers of the subject project, fails on at least one test. The developer-repaired version is a subsequent version of that code submitted by the project's developers to the project's version-control repository that passes all the tests, minimized to only include changes relevant to repairing the defect.

Defects4J has been used to evaluate program repair in terms of how often techniques produce patches [41], what types of defects the techniques are able to patch [98], and the quality of the produced patches [72], [90], [136], [137]. These existing evaluations that measure patch quality use manual inspection [72], [90], [136] or automatically-generated evaluation test suites [72], [135], [137]. While manual inspection is subjective and could be biased, low-quality evaluation test suites could inaccurately measure quality [72]. In this paper, we develop a methodology for producing high-quality evaluation test suites, allowing us to measure patch quality more accurately; we also go beyond simply measuring quality and study what factors influence patch quality of automated program repair.

4.2 Quality-Evaluating Test Suites

To objectively measure the quality of a generated repair, we need two independent test suites that specify the desired behavior of the program being repaired. One test suite can be used by the automated program repair techniques to produce a patch for a defect. The second, independent test suite is called the evaluation test suite; this test suite is used to measure the patch's quality. As already mentioned, each Defects4J defect comes with a developer-written test suite that evidences the defect. To create the second test suite, for each defect, we generated test inputs using an off-the-shelf automated test input generator, and using the developer-repaired code as an oracle of correct behavior. We generated the second test suites only for the 106 defects for which at least one of the four automated repair techniques we evaluate produced a patch. (Fig. 3 in Section 5.1 will describe these patch results.)

This repair-quality methodology is only effective if the evaluation test suite is of high-quality. Coverage is widely-used in industry to estimate test-suite quality [61]. Using statement-level code coverage as a proxy for test suite quality, our goal was to generate, for each defect, a high-coverage test suite, thus implying that a big portion of the functionality of the inspected class is being evaluated. Specifically, we focused on the statement coverage of the methods and classes modified by the developer-written patch and designed a test generation methodology aimed to maximize that coverage. Ideally, we want the evaluation test suite to have perfect coverage, but modern automated test generation tools cannot achieve perfect coverage on all large real-world programs, in part because of limitations of such tools such as possible infinite recursion in the creation process or impreciseness of method signatures such as Java generics [49]. Thus, we set as our goal to generate, for each defect, a test suite that achieves 100 percent coverage on all developer-modified methods, and at least 80 percent coverage on all developer-modified classes. The choice of coverage criteria is a compromise between a reasonable measure of covering all the developer changes and the modern automated test generation tools' ability to generate high-coverage test suites.

We used the patched version of the code to generate the evaluation test suite because it guarantees that this test suite covers at least one way of repairing the defect. An alternative to using the defective version of the code would not provide such a guarantee. Our choice might cause the evaluation test suites to more accurately measure the quality of patches that are structurally similar to the human-written patches, and would bias that measurement more favorably toward patches whose behavior agrees with the human-written patches. Future work could attempt to mitigate these concerns by combining test suites generated using multiple versions of the code, and by using alternate information for oracles, such as natural language specifications [17], [53], [97], [124], other implementations of the same specification [92], or even the unpatched version [135], [141], though each of those approaches would introduce its own limitations.

We compared the effectiveness of two modern off-the-shelf automated test generators Defects4J supports, Randoop [105] and EvoSuite [49], in a controlled fashion, and found that EvoSuite consistently produced test suites with higher coverage on Defects4J defects' code. This finding is consistent with prior analyses [118]. Accordingly, we elected to use EvoSuite as our test suite generator.

EvoSuite uses randomness in its test generation and continues to generate tests up to a given time budget, so we experimented with different ways to run EvoSuite to maximize coverage. We ran EvoSuite using branch coverage as its target maximization search criterion (the default option) twenty times per defect, with different seeds, ten times for 3 minutes and ten times for 30 minutes. We found low variance in the coverage produced by the generated test suites: the 3-minute test suites had a variance in statement coverage of 0.6 percent and the 30-minute test suites of 0.8 percent. We also found that the improvement between the mean statement coverage of the 3-minute test suites and the mean statement coverage of the 30-minute test suites was low (from 68 to 72 percent), suggesting that longer time budgets would not significantly improve coverage. Merging ten 3-minute test suites resulted in higher statement coverage than a single average 30-minute test suite (77 versus 72 percent). Finally, merging ten 30-minute test suites resulted in 81 percent statement coverage, on average, the highest we observed. We thus used the ten merged 30-minute test suites as preferred combination mechanism to optimize test suite coverage.

We followed the following automated process for generating the test suites: For each defect, we ran EvoSuite (v1.0.3) ten times (on different seeds) with a 30-minute time budget and merged the ten resulting test suites, removing duplicate tests. We then checked if the resulting test suite covered 100 percent of the statements in the developer-modified methods, and at least 80 percent of the statements in each of the developer-modified classes. For 34 out of the 106 defects, this algorithm generated test suites that satisfied the coverage criterion. In the course of our study, a new version of EvoSuite was released. We attempted to augment the test suites by

defect set	**# of defects**	**statement coverage of patch-modified**	**mean**	**median**
at least one patch	106	methods	90.8%	100.0%
		classes	87.2%	96.3%
adequate test suite	71	methods	100.0%	100.0%
		classes	96.7%	98.7%

Fig. 2. Statement coverage of the EvoSuite-generated test suites for the 106 Defects4J defects patched by at least one repair technique in our study, and for the 71-defect subset for which our generated test suites covered 100 percent of all developer-modified methods and at least 80 percent of all developer-modified classes.

using this later version of EvoSuite (v1.0.6), but this new version did not produce better-coverage test suites than v1.0.3 on its own. However, using statement-coverage as the target maximization search criterion (instead of the default branch coverage) did produce test suites that, when combined with the previous v1.0.3-generated test suites, improved coverage. This process resulted in test suites that satisfied the coverage criterion for a total of 62 defects (11 Chart, 6 Closure, 11 Lang, 30 Math, and 4 Time defects).

We then examined the generated test suites that met one, but not both of the coverage criteria and attempted to manually augment them to fully meet the other criterion. Examining these cases, we found that EvoSuite often was unable to cover statements that required the use of specific hard-to-generate literals present in the code. For example, covering some portions of code from the Closure project (a JavaScript compiler) required tests that take as input specific strings of JavaScript source code, such as an inline comment. Meanwhile covering some exceptional Lang code required specific strings to trigger the exceptions. The probability of the random strings generated and selected by EvoSuite to match the necessary strings to cover these portions of the code is negligibly small. We, therefore, manually examined the source code and created test cases using the necessary literals. Augmenting the EvoSuite-generated test suites with these manually-written tests resulted in high quality test suites for 9 more defects (1 Chart, 3 Closure, 4 Lang, and 2 Math, defects) that satisfied the coverage criteria.

In total, this process produced test suites that satisfied the coverage criterion for 71 of the 106 defects (12 Chart, 9 Closure, 14 Lang, 32 Math, and 4 Time defects). The test suites varied in size from 59 to 7,164 tests, with the mean test suite containing 1,194 tests and the median test suite 648 tests.

We restrict our study to these 71 defects. An additional 5 defects had 80 percent or higher coverage on the developer-modified classes, but did not have 100 percent coverage on the developer-modified methods. The mean statement coverage for the developer-modified classes for these 71 defects is 96.7 percent and the median is 98.7 percent (with means and medians for the modified methods both 100 percent, as required by the coverage criterion). Fig. 2 summarizes these statistics for the 71 defects used in our study and the 106 defects patched by at least one repair technique.

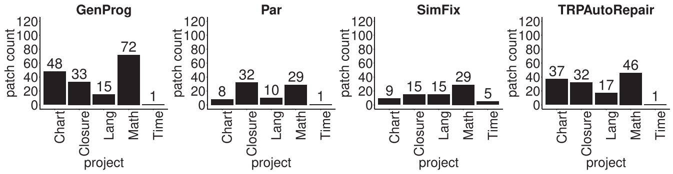
We examined the 35 defects for which our process failed to generate adequate test suites to understand why this happened. We found that the uncovered code was either unreachable, the default code at the end of a switch statement, a branch of a complex set of nested if statements, exception declarations or catch clauses for exceptions not thrown by local code (but possibly thrown elsewhere). Unfortunately, because significant domain knowledge and project-specific understanding are necessary to determine whether such code is reachable and to construct an input that would execute this code, we could not definitively eliminate it as unreachable, and elected to omit these defects from our study.

5 Empirical Measurements of Repair Quality

We evaluate *G&V* repair via a series of controlled experiments using the Defects4J dataset described in Section 4.1 and test suites described in Section 4.2. Section 5.1 outlines our experimental procedure for repairing defects using GenProg, Par, SimFix, and TrpAutoRepair and reports how successful the techniques are at producing patches on real-world defects. Section 5.2 examines the quality of those patches and measures which factors affect patch quality. Finally, Section 5.3 explores methods for improving patch quality.

technique	patches total	patches unique	defects patched
GenProg	585 (8.2%)	255	49 (13.7%)
Par	288 (4.0%)	107	38 (10.6%)
SimFix	76 (21.3%)	73	68 (19.0%)
TRPAutoRepair	513 (7.2%)	199	44 (12.3%)
total	1,462 (6.7%)	634	106 (29.7%)

(a) Produced patches



(b) Unique patch distributions, per technique

Fig. 3. (a) GenProg, Par, SimFix, and TrpAutoRepair produce patches 1,462 times (6.7 percent) out of the 21,777 attempts. At least one technique can produce a patch for 106 (29.7 percent) of the 357 real-world defects. (b) The distributions of unique patches produced by the four techniques are similarly shaped.

5.1 Ability to Produce a Patch

> *Research Question 1:* Do *G&V* techniques produce patches for real-world Java defects?

We used each repair technique to attempt to repair each of the 357 defects in the Defects4J benchmark providing the developer-written test suite to all the techniques to guide repair. For GenProg, Par, and TrpAutoRepair, which select random mutation operators to generate a patch, we attempt to repair each defect 20 times with a timeout of 4 hours each time, using a different seed each time, for a total of $357 \times 20 = 7{,}140$ attempted repairs, per repair technique. For SimFix, which is deterministic, we attempt the repair once for each defect using the default timeout of 5 hours, for a total of 357 attempted repairs. This results in a grand total of $7{,}140 \times 3 + 357 = 21{,}777$ repair attempts. We ran these techniques using a cluster of 50 compute nodes, each with a Xeon E5-2680 v4 CPU with 28 cores (2 processors, 14 cores each) running at 2.40 GHz. Each node had 128 GB of RAM and 200 GB of local SSD disk. We launched multiple repair attempts in parallel, each requesting 2 cores on one compute node. The 20 repair attempts provided a compromise between the likely ability to make statistically significant findings, and the computational resources necessary to run our experiments. The computational requirements are significant: Repairing a single defect 20 times with a 4-hour timeout can take 80 hours per defect per repair technique, and 10 CPU-years for 357 defects and 3 repair techniques.

The repair techniques' parameters affect how they attempt to repair defects. For GenProg, Par, and TrpAutoRepair (implemented in JaRFly), we used the parameters from prior work that evaluates these techniques on C programs [69], [77], [111]. We set the population size (`PopSize`) to 40 and the maximum number of generations to 10 for all three techniques. For GenProg and TrpAutoRepair, we uniformly equally weighted the mutation operators `append`, `replace`, and `delete`. For Par, we uniformly equally weighted the mutation operators `FUNREP`, `PARREP`, `PARADD`, `PARREM`, `EXPREP`, `EXPADD`, `EXPREM`, `NULLCHECK`, `OBJINIT`, `RANGECHECK`, `SIZECHECK`, and `CASTCHECK`. For GenProg and Par, we set `SampleFit` to 10 percent of the test suite. For fault localization, all three techniques apply a simple weighting scheme to assign values to statements based on their execution by passing and failing tests. For Par and TrpAutoRepair, we set `negativePath-Weight` to 1.0 and `positivePathWeight` to 0.1, based on prior work [69], [111]. For GenProg, we set `negativePath-Weight` to 0.35 and `positivePathWeight` to 0.65 [78]. For all remaining parameters, we use their default values from prior work [69], [77], [111]. For SimFix, we use its open-source implementation with its default configuration [62].

We describe the complete set of parameters at `https://github.com/LASER-UMASS/JavaRepair-replication-package/wiki/Configuration-parameter-details/`.

Fig. 3a reports the results of the repair attempts. GenProg patches 49 out of 357 defects (6 Chart, 15 Closure, 9 Lang, 18 Math, and 1 Time) and produces a total of 585 patches, out of which 255 are unique. Par patches 38 out of 357 defects (3 Chart, 12 Closure, 7 Lang, 15 Math, and 1 Time), and produces a total of 288 patches, out of which 107 are unique. SimFix patches 68 out of 357 defects (8 Chart, 15 Closure, 13 Lang, 27 Math, and 5 Time) and produces a total of 76 patches, out of which 73 are unique. TrpAutoRepair patches 44 out of 357 defects (7 Chart, 12 Closure, 8 Lang, 16 Math, and 1 Time) and produces a total of 513 patches, out of which 199 are unique. Overall, at least one technique produced at least one patch for 106 out of the 357 defects. All techniques produced at least one patch for 12 defects. SimFix most often produced patches (21.3 percent of the attempts) and produced patches for the most defects (19.0 percent). Fig. 3b shows the distributions of unique patches, per project, generated by each of the four techniques.

Compared to prior studies on C defects [122], [79], [111], the Java repair mechanisms produce patches on fewer repair attempts and for fewer defects. On C defects, GenProg produced patches for between 47 percent (ManyBugs defect dataset) and 60 percent (IntroClass defect dataset) and TrpAutoRepair produced patches for between 52 percent (ManyBugs) and 57 percent (IntroClass) defects. It is not surprising that on real-world defects, the rate is lower. Our findings are also consistent with prior work applying *G&V* repair to Java defects, which found techniques to produce patches for 9.8–15.6 percent of the defects [90]. In a prior study on Java defects, Par produced patches for 22.7 percent of the defects [69]. While that study's defects also came from real-world software projects, it is possible that the complexity of Defects4J defects results in the lower patch rates for Par. Some of the prior study's defects came from Lang and Math, projects that are also part of Defects4J (though a different set of defects), and our results on those projects are similar to those in the prior study [69]. Even though SimFix patches more defects (19.0 percent) than other techniques, the fraction of defects patched by SimFix is still much lower (19.0 versus 47 percent) than that those obtained using repair techniques for C defects.

Answer to Research Question 1: We conclude that *G&V* techniques do produce patches on real-world Java defects, though the rate of patch production is lower than on C defects.

5.2 Patch Quality

Section 5.1 showed that *G&V* techniques are able to patch 29.7 percent of the real-world defects in Defects4J. This section explores the quality of the produced patches and measures the factors that affect it. These experiments are based on the 71 defects for which we are able to generate high-quality evaluation test suites (recall Section 4.2). These 71 defects are a subset of the 106 defects for which at least one repair technique produced at least one patch (recall Fig. 2).

5.2.1 Patch Overfitting

Research Question 2: How often and how much do the patches produced by *G&V* techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification?

Methodology. To measure the quality of a produced patch, we start with the defective code version, apply the patch to that code, and execute the generated evaluation test suite. We call the total number of tests executed in the evaluation test suite T_{total} and the number of tests the patched version passes T_{pass}. The quality of a patch is $\frac{T_{pass}}{T_{total}}$, as defined by prior work [122]. A patch that passes all the tests in the evaluation test suite has 100 percent patch quality.

We also measure the quality of the defective code version by executing the evaluation test suite prior to applying the patch. This allows us to identify the quality improvement due to the patch.

Results. First, we consider the quality of the patches automated program repair techniques produce. Fig. 4 shows the distributions of the quality of the patches produced by each

technique	patch quality minimum	mean	median	maximum	100%-quality patches
GenProg	64.8%	95.7%	98.4%	100.0%	24.3%
Par	64.8%	96.1%	98.5%	100.0%	13.8%
SimFix	65.0%	96.3%	99.9%	100.0%	46.1%
TrpAutoRepair	64.8%	96.4%	98.4%	100.0%	19.5%

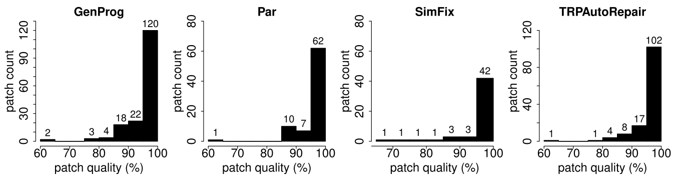
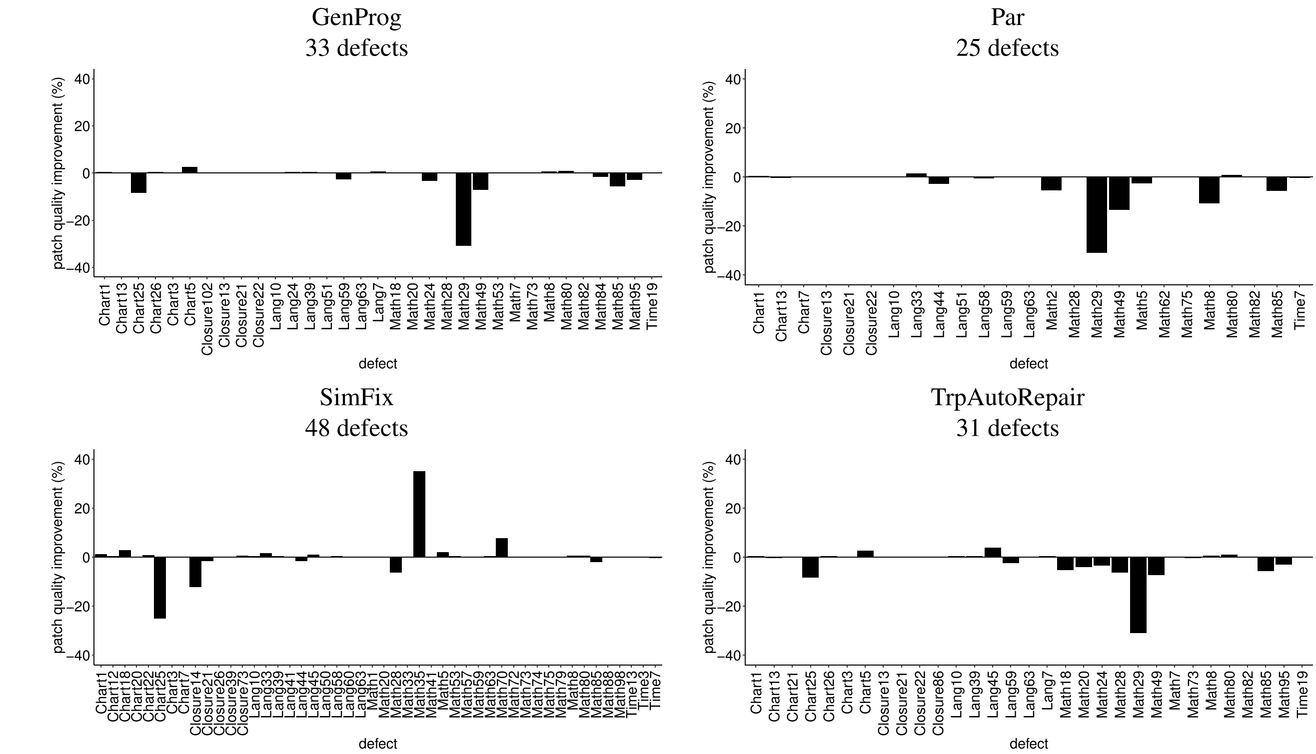


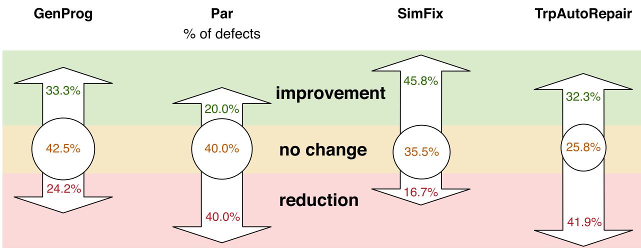
Fig. 4. The quality of the patches the repair techniques generated when using the developer-written test suite varied from 64.8 to 100.0 percent. The distributions of patch quality is skewed toward the 100 percent end. On average, 74.1 percent (GenProg: 75.7 percent, Par: 86.2 percent, SimFix: 53.9 percent and Trp: 80.5 percent) of the patches failed at least one test.

technique. Due to the nature of the space of possible patches, all four techniques produce the same patch for some defects, which, for example, caused the minimum exhibited quality patch to be identical for all four techniques. Overall, 74.1 percent of the patches (GenProg: 75.7 percent, Par: 86.2 percent, SimFix: 53.9 percent, and TrpAutoRepair: 80.5 percent), on average, failed at least one test, thus overfitting to the specification and failing to fully repair the defect. The mean quality of the patches varied from 95.7 to 96.4 percent. The relatively high fraction is not necessarily a proportional indication of the quality of repair: Defective code versions already pass 98.3 percent of the tests, on average, so a patch that passes 96.0 percent of the tests may not even be an improvement over the defective version.

Accordingly, next, we consider whether patches improve program quality. Fig. 5 shows, for each of the patched defects, the change in the quality between the defective version and the patched version. A negative value implies that the patched version failed more evaluation tests than the defective version. When a technique produced multiple distinct patches for a defect, for this comparison, we used the highest-quality patch. For GenProg, 33.3 percent of the defects' patches improved the quality, 42.5 percent showed no improvement, and the remaining 24.2 percent decreased quality. For Par, 20.0 percent improved, 40.0 percent showed no improvement, and 40.0 percent decreased quality. For SimFix, 45.8 percent improved, 35.5 percent showed no improvement, and 16.7 percent decreased quality. For TrpAutoRepair, 32.3 percent improved, 25.8 percent showed no improvement, and 41.9 percent decreased quality. For Par and TrpAutoRepair, more patches broke behavior than repaired it, and the decrease in quality was, on average, larger than the improvement. For all the techniques, the majority (89 out of 137, 65.0 percent) of the patches decrease or fail to improve quality, and more than a quarter (39 out of 137, 28.5 percent) of the patches break even more tests than they fix.

These results are consistent with the previous findings obtained using C repair techniques on small programs, where the median GenProg patch passed only 75 percent (mean 68.7 percent) of the evaluation test suite and the median TrpAutoRepair patch passed 75.0 percent of the evaluation test suite (mean 72.1 percent) [122].





technique	change in quality due to patch minimum	mean	median	maximum
GenProg	−30.9%	−1.7%	0.0%	2.6%
Par	−30.9%	−2.8%	0.0%	1.5%
SimFix	−24.9%	0.2%	0.0%	35.0%
TrpAutoRepair	−30.9%	−2.1%	0.0%	3.8%

Fig. 5. Patch overfitting. Change in quality between the defective version and the patched version of the code. The median patch neither improves nor decreases quality. While more GenProg patches improve the quality than decrease it, the opposite is true for Par and TrpAutoRepair patches, and, on average, patches break more functionality than they repair. The data presented are for the 45 defects with high-quality evaluation test suites, of which GenProg produced patches for 33, Par for 25, and TrpAutoRepair for 31.

Answer to Research Question 2: We conclude that tool-generated patches on real-world Java defects often overfit to the test suite used in constructing the patch, often breaking more functionality than they repair.

5.2.2 Test Suite Coverage and Size

Research Question 3: How do the coverage and size of the test suite used to produce the patch affect patch quality?

Intuition suggests that higher coverage test suites used to produce patches should lead to better-quality patches. Prior work empirically supports this intuition for *G&V* program repair [122]; however, that work approximated the test suite coverage using test suite size and was not on real-world defects. In this study, we use real-world defects, measure the actual statement-level code coverage instead of an estimate or proxy, and control for confounding factors, such as test suite size, defects' project, and the number of failing tests. In fact, prior studies of test suites have identified test suite size as often a confounding factor [67]. For our dataset, we found statistically significant weak positive correlation ($r = 0.14$) between test suite size and statement-level coverage of the developer-written test suite on the defective code version. This is consistent with the prior studies [67].

Methodology. To measure the relationship between test suite coverage and repair quality, we attempted to create subsets of the developer-written test suite of varying coverage while controlling for test suite size, number of failing tests, and the defects themselves. However, we found that there is very low variability in the coverage of the individual tests and so we could not control for the test suite size while varying coverage. Hence, we generate the subsets while controlling for the number of failing tests and defects. Since test suite coverage and test suite size are positively correlated, analyzing their association with repair quality individually would not be appropriate. Thus, we use multiple linear regression to identify the relationship between two explanatory variables (test suite coverage and test suite size) and a response variable (repair quality). Unlike prior work [122], our methodology does not need to control for the ratio of passing to failing tests because most of the Defects4J defects have only a single failing test. (Section 5.2.3 will discuss the lack of variability in the number of failing tests further.)

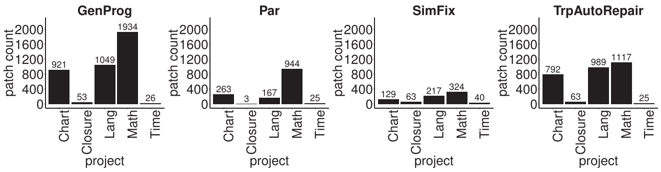
For this analysis, we considered the 71 defects for which we created high-quality evaluation test suites. For each of the defects, we created subsets of the developer-written test suite of varying coverage. Each subset contains all the tests that evidence the defect, and randomly selected subsets of the rest of the tests. We then used the repair techniques to produce patches using these test suite subsets (using the methodology from Section 5.1), and then computed the quality of the patches produced for each defect using the automatically-generated evaluation test suites. We excluded defects for which we could not generate test suites with sufficient variability in coverage, and, as before, for which we did not have sufficiently high-quality evaluation test suites. We describe the details of our methodology next.

To generate the test suite subsets for each defect, we first compute the minimum and the maximum code coverage ratio of the developer-written test suite of that defect. The *minimum code coverage ratio* ($cov_{\min}$) of a developer-written test suite is the statement coverage on the defective code version of just those tests that fail on the defective code version and pass on the developer-repaired code version. We include all of these tests in every subset we generate, so their coverage is the minimum possible coverage. The *maximum code coverage ratio* ($cov_{\max}$) is the statement coverage on the defective code version of the entire developer-written test suite (the largest possible subset). For example, for Chart 1, there is 1 failing test and 245 passing tests that execute the developer-modified class `AbstractCategoryItemRenderer`. The minimum coverage, ($cov_{\min}$), for Chart 1 is the statement coverage of the single failing test on the developer-modified class. This test covers 18 out of the 519 lines, (3.5 percent). The maximum coverage, ($cov_{\max}$), for Chart 1 is the statement coverage of the full test suite (246 tests) on the developer-modified class. This test suite covers 300 out of the 519 lines, (57.8 percent).

We then compute the potential test suite coverage variability as the difference between the minimum and the maximum: $\Delta_{cov} = cov_{\max} - cov_{\min}$. Defects whose $\Delta_{cov} < 25\%$ lack sufficient variability in statement coverage to be used in this study, and we discard them. In our study, we discarded 15 defects for this reason (2 Chart, 1 Closure, 1 Lang and 11 Math) out of the 71 defects that had at least one repair technique produce at least one patch and had a high-quality evaluation test suite (recall Section 4.2).

For each of the 56 remaining defects, we chose five target coverage ratios evenly spaced between the minimum and the maximum: $cov_{\min} + \frac{1}{5}\Delta_{cov}$, $cov_{\min} + \frac{2}{5}\Delta_{cov}$, $cov_{\min} + \frac{3}{5}\Delta_{cov}$, $cov_{\min} + \frac{4}{5}\Delta_{cov}$, and $cov_{\min} + \Delta_{cov} = cov_{\max}$.

We used these target ratios to create 25 distinct test suites, 5 for each of the targets. For each target ratio c, we attempted to create five distinct test suite subsets within a 5 percent margin of c. (Note that there are typically multiple ways to achieve even $cov_{\max}$ coverage.) Each of the five test suite subsets started with all tests that fail on the defective code version and pass on the developer-repaired code version. We then iteratively attempted to add a uniformly randomly selected passing test case, without replacement, one



(a) Distribution of patches generated using varying-coverage test suites.

technique	minimum	patch quality mean	median	maximum	100%-quality patches
GenProg	0.0%	94.8%	98.4%	100.0%	16.2%
Par	51.8%	91.2%	95.5%	100.0%	13.3%
SimFix	77.3%	98.4%	100.0%	100.0%	50.7%
TrpAutoRepair	62.9%	95.5%	99.0%	100.0%	19.0%

(b) Quality of patches generated using varying-coverage test suites.

technique	model quality p	R^2	test suite	p
GenProg	7.2×10^{-13}	0.013	size	6.7×10^{-13}
			coverage	8.5×10^{-4}
Par	5.2×10^{-12}	0.035	size	4.2×10^{-5}
			coverage	7.6×10^{-11}
SimFix	4.0×10^{-16}	0.086	size	2.7×10^{-7}
			coverage	1.3×10^{-15}
TrpAutoRepair	6.9×10^{-5}	0.0057	size	1.6×10^{-5}
			coverage	0.96

(c) Multiple linear regression relating coverage and size to patch quality.

Fig. 6. Test suite coverage and size. (a) Distribution of the number of patches produced using developer-written test suite subsets of varying code coverage on the defective code version. (b) The quality of the patches generated using varying-coverage test suites varied from 0.0 to 100.0 percent. On average, 75.2 percent (GenProg: 83.8 percent, Par: 86.7 percent, SimFix: 49.3 percent, and TrpAutoRepair: 81.0 percent) of the patches failed at least one test. (c) A multiple linear regression reports that test suite size and test suite coverage are strongly significantly associated with patch quality ($p < 0.001$) except for coverage for TrpAutoRepair).

at a time, as long as it did not make the subset's coverage exceed the target by more than 5 percent, stopping if the subset's coverage was within 5 percent of the target. If we attempted to add a test 500 times and failed to reach the target, we stopped. For 11 of the 56 defects (2 Chart, 3 Closure, 1 Lang, and 5 Math), the sampling algorithm was unable to generate five distinct test suite subsets for all of the targets, so we discard these 11 defects. We consider the remaining 45 defects for the analysis.

Finally, for each technique, we computed a multiple linear regression considering patch quality as the dependent variable and test suite coverage and size as independent variables.

Results. For each of the 45 defects, we had 25 test suite subsets, and we attempted each repair 20 times using GenProg, Par, and TrpAutoRepair on different seeds, and one time using SimFix. In total, these 23,625 repair attempts produced 9,144 patches. Fig. 6a shows the distribution of these patches. GenProg produced at least one patch for 29 out of the 45 defects, Par 25, SimFix 34, and TrpAutoRepair 29. (GenProg: 6 Chart, 2 Closure, 10 Lang, 10 Math, and, 1 Time; Par: 5 Chart, 1 Closure, 8 Lang, 10 Math, and, 1 Time; SimFix: 6 Chart, 3 Closure, 8 Lang, 13 Math, and 4 Time; and TrpAutoRepair 6 Chart, 2 Closure, 10 Lang, 10 Math, and, 1 Time.)

Fig. 6b shows the statistics of the quality of the patches for those defects, created using the varying-coverage test suites. The quality varied, with GenProg even producing some patches that failed *all* evaluation test cases. Overall, 75.2 percent of the patches, on average, failed at least one test in the evaluation test suite.

Next, for each technique, we created a multiple linear regression model to predict the quality of the patches based on the test suite coverage and size. Fig. 6c shows, for each technique, the results of the regression model. All four fitted regression models are strongly statistically significant ($p < 0.001$) though with low R^2 values. Test suite size was a statistically significant predictor for patch quality for all four techniques, with larger test suites leading to higher-quality patches; however, with an extremely small effect size. Coverage was a less clear predictor: for TrpAutoRepair, the association was not statistically significant ($p > 0.1$), and was positive for GenProg and TrpAutoRepair, but negative for SimFix and Par. We further detail each technique's regression results next.

For GenProg, patch quality (on a 0–100 scale) is equal to $94.82 - 0.02(\text{coverage}) + 0.02(\text{size})$, where coverage is $100\times$ the fraction of code in the defective code version covered by the test suite, and size is the normalized number of tests in the test suite used to generate the patch. Thus, the quality of the patch produced by GenProg decreases by 0.02 percent for each 1 percent increase in the test suite coverage and increases by 0.02 percent for each additional test in the test suite. While both associations of test suite coverage and size with the patch quality were statistically significant ($p < 0.001$), the magnitude is extremely small and the low R^2 value indicates little of the variability is explained. We conclude that test suite coverage and test suite size are significant predictors of patch quality, but the magnitude of the effect is extremely small, for GenProg.

For Par, the quality of the patch is equal to $91.18 - 0.10(\text{coverage}) + 0.03(\text{size})$. Thus, the quality of the patch produced by Par decreases by 0.10 percent for each 1 percent increase in the test suite coverage and increases by 0.03 percent for each additional test in the test suite. Again, while both associations of test suite coverage and test suite size with patch quality are strongly statistically significant ($p < 0.001$), the magnitude is extremely small and the low R^2 value indicates little of the variability is explained. We conclude that both test suite coverage and test suite size are significant predictors of patch quality, but the magnitude of the effect is extremely small, for Par.

For SimFix, the quality of the patch is equal to $98.43 - 0.04(\text{coverage}) + 0.002(\text{size})$. Thus, the quality of the patch produced by SimFix decreases by 0.04 percent for each 1 percent increase in the test suite coverage and increases by 0.002 percent for each additional test in the test suite. We observe strongly statistically significant ($p < 0.001$) associations of test suite coverage and test suite size with patch quality however, the magnitude is extremely small and the low R^2 value indicates little of the variability is explained. We conclude that both test suite coverage and test suite size are significant predictors of patch quality, but the magnitude of the effect is extremely small, for SimFix.

For TrpAutoRepair, the quality of the patch is equal to $95.80 + 0.0003(\text{coverage}) + 0.006(\text{size})$. The equation implies that the quality of the patch produced by TrpAutoRepair increases by 0.0003 percent for 1 percent increase in the test suite coverage and increases by 0.006 percent for each additional test in test suite. The association of test suite size with patch quality is strongly statistically significant ($p < 0.001$), but that is not the case for test suite coverage. And, again, the magnitude of the association is extremely small and the low R^2 value indicates little of the variability is explained. We conclude that test suite size is a significant predictor of patch quality, but the magnitude of the effect is extremely small, for TrpAutoRepair.

Answer to Research Question 3: We conclude that, surprisingly, both test suite size and test suite coverage have extremely small but statistically significant correlations with patch quality (positive for test suite size and negative for test suite coverage) produced using automatic program repair techniques.

Previous findings for C program repair techniques [122] considered only test suite size and found that for both GenProg and TrpAutoRepair, larger test suites improved patch quality.

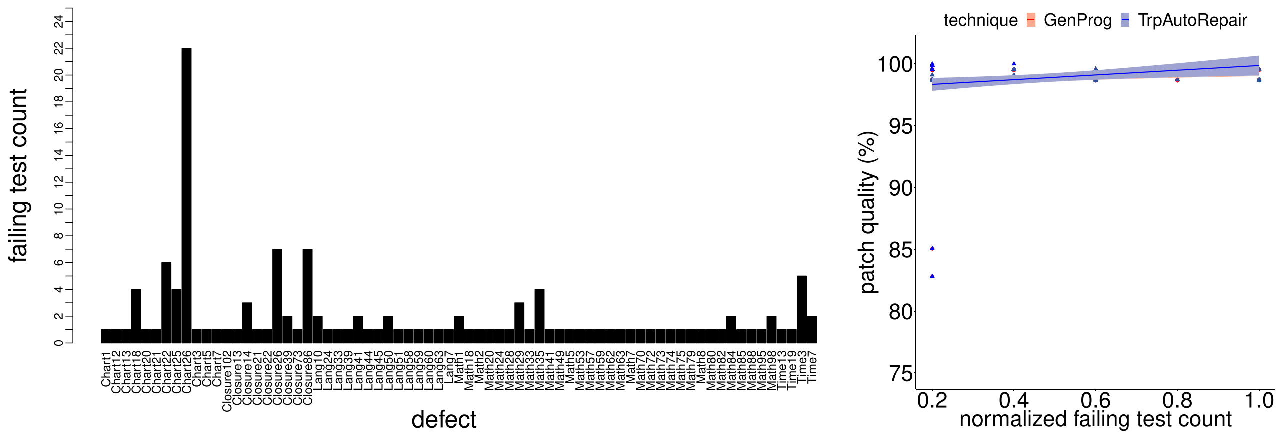
5.2.3 Defect Severity

Research Question 4: How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit?

The number of failing tests that trigger the defect are likely to be proportional to the number constraints that repair techniques need to satisfy to generate a repair. The goal of this research question is to measure the effect of the number of failing tests in the test suite used for producing the patches on the quality of patches generated using *G&V* techniques.

Methodology. To measure the effect of the number of failing tests in the test suite used to guide repair, we selected those defects that had at least 5 failing tests in the developer-written test suite and for which we are able to create high-quality evaluation test suite (recall Section 4.2). Unfortunately, there were only 5 such defects in the 71-defect subset of Defects4J. For each of the five defects, we created 21 test suites subsets. We did this by first computing five evenly distributed target sizes s: $\frac{1}{5}f$, $\frac{2}{5}f$, $\frac{3}{5}f$, $\frac{4}{5}f$, and f, where f is the number of failing tests in the developer-written test suite (rounding to the nearest integer). Then, for each s (except $s = f$), we created 5 test suite subsets by including every passing test from the developer-written test suite, and uniformly randomly sampling, without replacement, s of the failing tests. This created 20 test suite subsets. We also included the entire developer test suite as a representative of the $s = f$ target, for a total of 21 test suite subsets. We then used the four automated repair techniques to attempt to patch the defects using each of the test suite subsets, following the methodology described in Section 5.1. Our methodology controls for the number of passing tests, unlike the prior study [122].

Both patch quality and the number of failing tests in the test suite used to guide repair are continuous variables, so we measure the association between these two variables using the Pearson correlation coefficient. This is typical for



(a) (b)

Fig. 7. Defect severity. (a) The distribution of the number of failing tests in the 71 defects for which at least one repair technique produces at least one patch and has a high-quality evaluation test suite. (b) Linear regression between patch quality and the number of failing tests and Pearson's correlation show statistically significant positive correlations for GenProg and TrpAutoRepair.

measuring the linear relationship between two continuous random variables.

Results. Fig. 7a shows the frequency distribution of failing tests across the 71 defects for which at least one of the four techniques produced at least one patch, and for which we were able to create a high-quality evaluation test suite. Of these 71 defects, only 5 defects, Chart 22, Chart 26, Closure 26, Closure 86, and Time 3, have at least five failing tests.

Fig. 7b shows, for each technique, the quality of the patches produced, as a function of the fraction of the failing tests in the test suite used to guide repair. For GenProg and TrpAutoRepair, we observe statistically significant ($p < 0.05$) positive correlations (GenProg: $r = 0.18$, $p = 0.006$; TrpAutoRepair: $r = 0.19$ $p = 0.008$) between patch quality and the number of failing tests in the test suite. The 95 percent confidence interval for both techniques was $[0.05, 0.30]$.

Par did not produce any patches for any of the 5 defects considered for this analysis. Simfix only produced three patches and did not patch any of the 5 defects when using partial failing tests. Analyzing the execution logs of SimFix revealed that it was not able to localize the bug using partial failing tests. This suggests that fault localization strategy used by repair techniques could be a confounding factor when measuring the effect of the number of failing tests on patch quality. (Recall that SimFix and JaRFly use different fault localization techniques.)

> *Answer to Research Question 4:* We conclude that the number of tests that a buggy program fails has a small but statistically significant positive effect on the quality of the patches produced using automatic program repair techniques and that this finding depends on the fault localization strategy used by the repair techniques.

5.2.4 Test Suite Provenance

> *Research Question 5:* How does the test suite provenance (whether it is written by developers or generated automatically) influence patch quality?

Prior work has suggested that using automatic test generation might improve program repair quality by increasing the coverage of the test suite used to produce the repair [122], [135], [141]. Augmenting a developer-written test suite with automatically-generated tests requires an oracle that specifies the expected test outputs. The unpatched program can be used as that oracle [135], [141], but that enforces the assumption that the patch should avoid changing any behavior not explicitly exhibited by the failing tests. Other implementations of the same specification could similarly be used as an oracle [92], but this is only possible when multiple implementations exist (e.g., if repairing a browser and the expected behavior can be observed in an independent browser implementation) and requires defects in the implementations to be independent, which is often not the case in practice [70]. Finally, oracles can perhaps be extracted from comments or natural language specifications, for example with Swami [97], Toradacu [53], Jdoctor [17], or @tComment [124].

However, our earlier study found that even when a perfect oracle exists, using automatically-generated tests for program repair resulted in much lower quality patches than using developer-written tests (about 50 percent versus about 80 percent quality) on small, student-written programs [122]. Thus, this research question sets out to evaluate the effectiveness of using tests generated using EvoSuite as described in Section 4.2 to produce patches using *G&V* repair.

Methodology. In this experiment, we compared the patches generated using developer-written test suites from Section 5.1 to patches generated using the EvoSuite-generated test suites. A technical challenge in executing repair techniques using EvoSuite-generated tests is a potential incompatibility between the bytecode instrumentation of EvoSuite-generated tests with the bytecode instrumentation done by code-coverage-measuring tools employed by repair techniques for fault localization. JaRFly uses JaCoCo [59] for fault localization and resolves instrumentation conflicts by updating the runtime settings of EvoSuite-generated tests (following official EvoSuite documentation).[1] The EvoSuite-generated tests are compatible with

1. http://www.evosuite.org/documentation/measuring-code-coverage/

JaCoCo, Cobertura [27], Clover [8], and PIT [30] code coverage tools, but not with GZoltar [22]. Unfortunately, SimFix uses GZoltar, and so could not be included in this experiment. For GenProg, Par, and TrpAutoRepair, as before, we used the developer-written patches as the oracle of expected behavior.

To control for the differences in the defects, properly measuring the association between test suite provenance and patch quality should be done using defects that can be patched using both kinds of test suites. If the set of defects patched using developer-written test suites differs from the set of defects patched using the automatically-generated test suites (as was the case in the earlier study [122]), then the defects can be a confounding factor in the experiment. For example, it is possible that more of the defects patched using one of the types of test suites are easier to produce high-quality patches for, unfairly biasing the results.

We thus started with the 68 defects for which at least one of the three repair techniques (GenProg, Par, and TrpAutoRepair) was able to produce a patch when using the developer-written test suites to guide repair, and first discarded those defects for which the EvoSuite-generated test suites did not evidence the defect. To evidence the defect, at least one test in the test suite has to fail on the defective code version. (By definition, all automatically-generated tests pass on the developer-patched version, since that version is the oracle for those tests.) For 31 out of the 68 defects, automatically-generated test suites did not evidence the defect. This left 37 defects (5 Chart, 4 Closure, 11 Lang, 16 Math, and 1 Time). We next executed each of the three repair techniques on each of the 37 defects using the EvoSuite-generated test suites, using the methodology from Section 5.1, thus executing $37 \times 20 = 740$ repair attempts per technique. Note that comparing repair techniques' behavior with different test suites on these 37 defects is unfair because one of the criteria they satisfied to be selected is that at least one repair technique produced at least one patch for the defect using the developer-written test suite. Thus, for each technique, we identified the set of defects that were patched both using developer-written and using automatically-generated test suites. We call these the *in-common* populations. Note that these populations are, potentially, different for each technique.

To compare the quality of the patches on the in-common patch populations, we use the nonparametric Mann-Whitney U test. We choose this test because the two populations may not be from a normal distribution. This test measures the likelihood that the two populations came from the same underlying distribution. We compute Cliff's delta's δ estimate to capture the magnitude and direction of the estimated difference between the two populations. We also compute the 95 percent confidence interval (CI) of the δ estimate.

Results. Fig. 8 summarizes our results. Fig. 8a reports data for the 37 defects for which both test suites evidence the defect. As expected, because of the aforementioned bias in the selection of the 37 defects, using EvoSuite-generated test suites produced fewer patches and patches for fewer defects than using developer-written test suites. Using developer-written test suites produced a patch on between 10.1 and 21.4 percent executions, while using EvoSuite-generated test suites produced a patch on between 2.3 and 13.9 percent of the executions. Using developer-written test suites produced a patch for between 54.1 and 81.1 percent of the defects, while using EvoSuite-generated test suites produced a patch for between 5.4 and 45.9 percent of the defects.

In addition to the bias in defect selection, another possible reason that EvoSuite-generated test suites resulted in fewer patches could be differences in the test suites. Fig. 8b shows the distributions of the number of failing (defect-evidencing) tests across the 37 defects for the two types of test suites. EvoSuite-generated test suites typically had more failing tests, perhaps contributing to it being more difficult to produce patches when using those test suites. Prior work has shown that having a larger number of failing tests correlated with lower patch production [98], [122].

We compared the quality of the patches produced using the two types of test suites on the in-common populations. Fig. 8c shows that for GenProg and TrpAutoRepair, the mean and median quality of the patches produced using the developer-written test suites are higher than of those produced using EvoSuite-generated test suites. These differences are statistically significant (Mann-Whitney U test, $p = 1.3 \times 10^{-11}$ for GenProg, and $p = 5.8 \times 10^{-11}$ for TrpAutoRepair). The δ estimate computed using Cliff's delta shows a large effect size for the median patch quality of the patches produced using EvoSuite-generated test suites being lower for GenProg and TrpAutoRepair. The 95 percent CI does not spans 0 for both techniques, indicating that, with 95 percent probability, the two populations are likely to have different distributions.

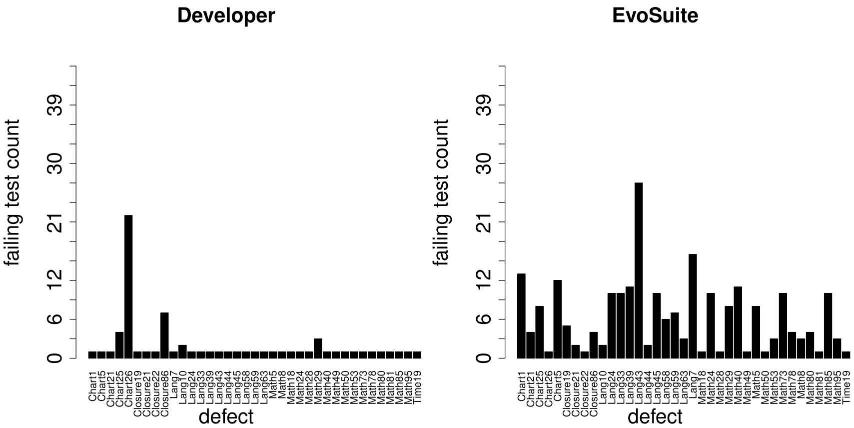
For GenProg, this comparison is on the 12 in-common defects (Chart 5, Closure 22, Lang 43, Math 24, Math 40, Math 49, Math 50, Math 53, Math 73, Math 80, Math 81, and Time 19). On these defects, GenProg produced 73 patches using developer-written test suites and 93 patches using EvoSuite-generated test suites (166 patches total). For TrpAutoRepair, this comparison is on the 13 in-common defects (Chart 5, Closure 22, Closure 86, Lang 43, Lang 45, Math 24, Math 40, Math 49, Math 50, Math 73, Math 80, Math 81, and Time 19). On these defects, TrpAutoRepair produced 57 patches using developer-written test suites and 96 patches using EvoSuite-generated test suites (153 patches total).

Because the results for GenProg and TrpAutoRepair are derived from 12 and 13 defects, respectively, there is hope that these results will generalize to other defects. The same cannot be said for Par. Par produced patches using both types of test suites for only 2 out of the 37 defects (Closure 22 and Math 50). Fig. 8c shows that the mean and median quality of the patches produced using the developer-written test suites are lower than those produced using EvoSuite-generated test suites. This result is statistically significant because Par produced 18 patches using developer-written test suites and 17 patches using EvoSuite-generated test suites, with $p = 5.3 \times 10^{-5}$ and the 95 percent CI interval does not span 0. However, while significant for these 2 defects, we cannot claim (nor do we believe that) this result generalizes to all defects from this 2-defect sample.

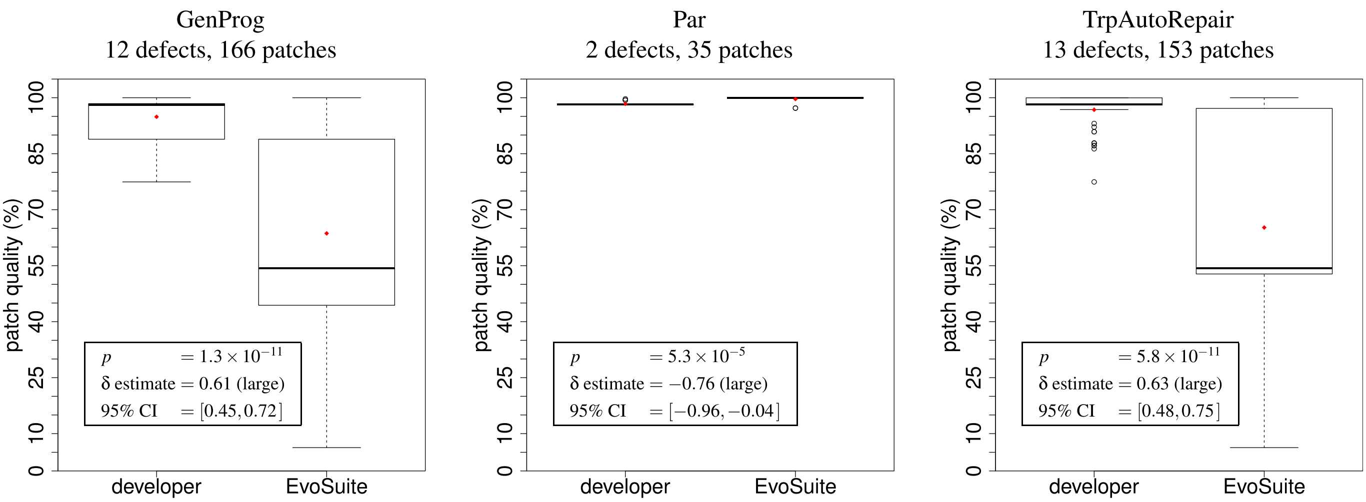
Our finding is consistent with the earlier finding [122] that provenance has a significant effect on repair quality, and that for GenProg and TrpAutoRepair, developer-written test suites lead to higher quality pathces. Surprisingly,

technique	test suite	generated patches	defects patched	patch quality minimum	patch quality mean	patch quality median	patch quality maximum	100%-quality patches
GenProg	developer	158 (21.4%)	29 (78.4%)	77.4%	94.9%	98.0%	100.0%	17.8%
	EvoSuite	98 (13.2%)	14 (37.8%)	6.3%	65.3%	54.3%	100.0%	8.2%
Par	developer	75 (10.1%)	20 (54.1%)	98.1%	98.4%	98.1%	99.7%	0.0%
	EvoSuite	17 (2.3%)	2 (5.4%)	97.2%	99.6%	99.9%	100.0%	41.2%
TrpAutoRepair	developer	128 (17.3%)	30 (81.1%)	77.4%	96.8%	98.1%	100.0%	24.6%
	EvoSuite	103 (13.9%)	17 (45.9%)	6.3%	65.2%	54.3%	100.0%	10.4%

(a) Patching results for the 37 Defects4J defects whose developer-written and EvoSuite-generated test suites have at least one failing test each.



(b) Distributions of failing tests in the 37 Defects4J defects' test suites.



(c) Patch quality comparison on the in-common (patched using both types of test suites) defect populations.

Fig. 8. Test suite provenance. (a) Using EvoSuite-generated test suites, automated program repair techniques were able to produce patches for 37 of the the 68 defects. (b) The EvoSuite-generated test suites typically have more failing tests than the developer-written ones. (c) The box-and-whisker plots compare patch quality on the in-common defect populations, showing the maximum, top quartile, median, bottom quartile, and minimum values, with the mean as a red diamond. The quality of patches produced by GenProg and TrpAutoRepair using the EvoSuite-generated test suites is statistically significantly (Mann-Whitney U test) lower that those produced using developer-written test suites. For Par, the effect is reversed.

the finding is opposite for Par (which was not part of the earlier study), with automatically-generated tests leading to higher-quality patches. Our study improves on the earlier work in many ways: We control for the defects in the two populations being compared, we use real-world defects, and we use a state-of-the-art test suite generator with a rigorous test suite generation methodology. The earlier study used a different generator (KLEE [21]) and aimed to achieve 100 percent code coverage on a reference implementation, but the generated test suites were small.

Answer to Research Question 5: We conclude that test suite provenance has a significant effect on repair quality, though the effect may differ for different techniques. For GenProg and TrpAutoRepair, patches created using automatically-generated tests had lower quality than those created using developer-written test suites. For a small, perhaps non-representative number of defects, Par-generated patches showed the opposite effect.

5.3 Mitigating Overfitting

> *Research Question 6:* Can overfitting be mitigated by exploiting randomness in the repair process? Do different random seeds overfit in different ways?

Because automated program repair aims to solve an underspecified problem, there are often many possible patches. This is the fundamental issue behind the repair quality problem. The partial specification — a test suite — fails to distinguish between patches that pass the tests and implement the desired functionality and the patches that pass the tests but fail to implement the desired functionality not encoded by the partial specification. The search space of possible patches is large [87] and navigating it in a way to improve the probability of finding a high-quality patch [68], [87], [88], [135] is at the heart of solving the repair quality problem.

An interesting observation is that the diversity of the patches produced in such a way, even by a single technique, may be used to improve the overall quality of a patch [122]. In essense, if each of the generated patches is wrong on the unspecified part of the specification, but is wrong in a different way, perhaps they can be combined in a way to produce a higher-quality patch. Specifically, a super patch that simulates the individual patches and then executes the plurality behavior may avoid the pitfalls of individual patches.

This is a form of n-version programming, and it is subject to the same constraints as n-version programming. Specifically, human program repair usually lacks the scale of diversity required to effectively combine programs into n-versions and meaningfully improve quality; correlations in faults of human-written programs prevent a quality improvement beyond some level [70]. Thus, testing if this approach works for automatically generated patches is, in some sense, a measure of whether human-written and automatically-generated patches differ in their diversity profiles.

Combining complex programs with side effects and potential resource use and contention, including simulating the execution of a set of patches in parallel, can be problematic. For this study, we separate the question of how to combine patches from the question of whether it might be worthwhile to combine patches. We answer the latter question. We simply say that if, given a set of patches for a defect, the majority of the patches passes an evaluation test, then it is possible that the n-version combination would pass that test. If the overall quality of an n-version patch across the entire evaluation test suite is higher than that of the individual patches, then perhaps it is worthwhile to attempt to combine them. Conversely, if the n-version patch quality is no better than the individual patches, combining is unlikely to improve quality.

Methodology. In Section 5.1, we described executing the four repair techniques on all 357 Defects4J defects using the developer-written test suites, with 20 different seeds per defect for GenProg, Par, and TrpAutoRepair, and once for SimFix. This produced 634 unique patches (255 by GenProg, 107 by Par, 73 by SimFix, and 199 by TrpAutoRepair, recall Fig. 3). For each technique, we identified the defects for which that technique produced at least 3 distinct patches. For these defects, we then evaluated how the potential n-version patch would perform by executing the evaluation test suite on each patch and considering the n-version to pass the test if the strict majority of the patches passed the test. For GenProg, 30 defects qualified for this experiment, 9 for Par, and 25 for TrpAutoRepair. SimFix could not be used for this analysis because it did not generate more than two distinct patches for any defect.

To compare the quality of the n-version and individual programs, we use the nonparametric Mann-Whitney U test. We choose this test because our data may not be from a normal distribution. We compute Cliff's delta's *delta estimate* to capture the magnitude and direction of the estimated difference between the two populations. We also compute the 95 percent confidence interval (CI) of the δ estimate.

Results. Fig. 9 compares the quality of the n-version patches to the individual patches that make up those n-version patches. The Mann-Whitney U test indicates the differences between the patch quality of the individual patches and the n-version patches are not statistically significant and the δ estimate suggests the differences are negligible.

> *Answer to Research Question 6:* We conclude that automated program repair techniques' patches lack the diversity necessary to employ an approach based on n-versioning to improve patch quality.

Our finding is consistent with the prior study for relatively high-quality patches [122]. However, the earlier study found that when patch quality was low (e.g., because of a low-quality test suite being used to repair the defect) the patch diversity may have been sufficient to improve quality [122]. This study does not explore that part of the question because the patches we observe for the Defects4J defects tend to be of relatively-high quality.

6 Discussion

Our main finding is that patches produced by Java *G&V* automated program repair techniques often overfit to the tests used to produce those patches. The most important implication of our work is that research is needed into improving program repair techniques to produce higher-quality patches, or at least identifying and discarding lower-quality ones. Researchers can use the patch quality evaluation methodology and high-quality test suites we have developed to evaluate their techniques on real-world defects and demonstrate improvements over the state-of-the-art within this important dimension.

We observed that test-suite size correlates with higher-quality patches, and test-suite coverage correlates with lower-quality patches, though both effects are extremely small. These findings, surprisingly, suggest that improving test suites used for repair is unlikely to lead to better patches. Future research should explore if there exists other guidance developers can use to improve their test suites to help program repair produce higher-quality patches.

Controlling for fault localization strategy, the number of tests a buggy program fails is positively correlated with higher-quality patches. On its face, this is surprising because fixing a larger number of failing tests usually requires fixing

technique	patch quality minimum	mean	median	maximum	100%-quality patches
GenProg	78.7%	96.7%	100.0%	100.0%	54.9%
GenProg (n-version)	75.8%	95.7%	99.9%	100.0%	50.0%
Par	82.4%	97.7%	100.0%	100.0%	76.5%
Par (n-version)	82.4%	97.6%	100.0%	100.0%	66.7%
TrpAutoRepair	80.1%	97.7%	100.0%	100.0%	59.3%
TrpAutoRepair (n-version)	75.8%	96.3%	100.0%	100.0%	56.0%

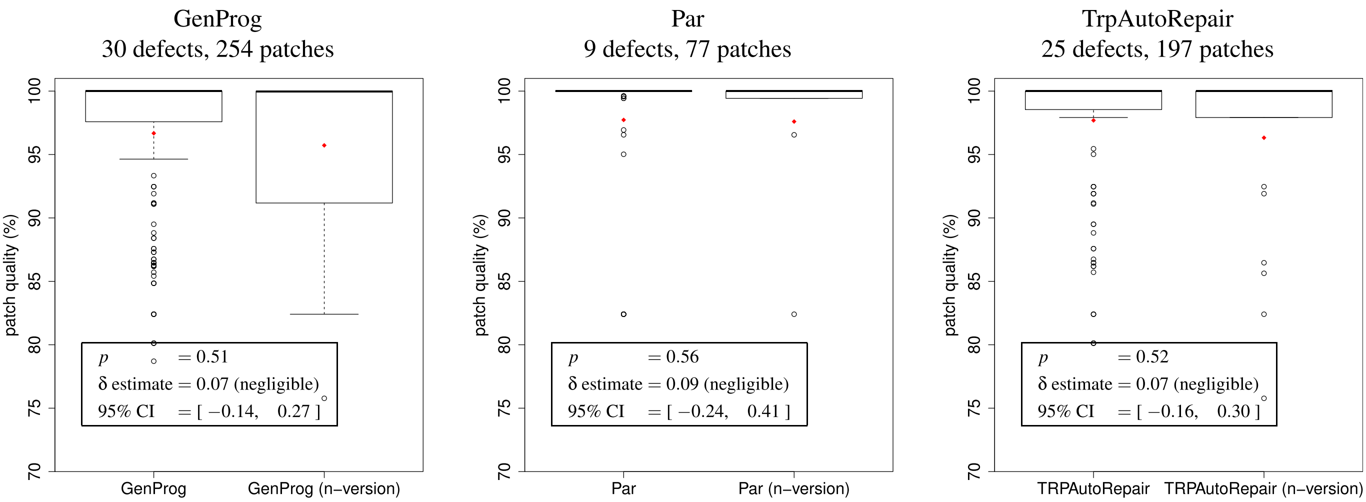


Fig. 9. The box-and-whisker plots compare the quality of the individual and n-version programs made up of those patches, with the mean as a red diamond. The p values (Mann-Whitney U test) suggest that there is no statistically significant difference in the quality of n-version and individual programs. We measure the effect size using Cliff's Delta test. For the given dataset, n-version programs perform negligibly worse (indicated by the δ estimate) than individual versions for all the three techniques however, the 95 percent confidence interval spans 0 for all techniques suggesting that, with 95 percent probability, the quality of n-version program is likely to be same as individual program.

more behavior (although it is certainly possible for a small bug to cause many tests to fail, and for a large bug to cause only one test to fail). The key observation here is that fault localization can be a confounding factor. A larger number of failing tests can help fault localization identify the correct place to repair a defect, improving the chances the technique can produce a patch. A recent study similarly found that fault localization can have a significant effect on repair quality [3]. In our study, we observe cases in which SimFix failed to localize a defect, and therefore failed to produce a patch when given fewer failing tests, but was able to do so with more failing tests (recall Section 5.2.3).

We found that human-written tests are, usually, better for program repair than automatically-generated ones. This suggests that automatically generating tests to augment the developer-written tests may not help program repair. However, the method of generating the tests likely matters, and future research should study that relationship, in particular, exploring whether new approaches that generate tests from natural-language specifications [17], [97] are helpful.

Finally, we observed that Java *G&V* repair techniques produce patches for more defects than C *G&V* repair techniques. Future research could target understanding the differences in the languages that cause this and improving the fix space and repair strategies used by the Java repair techniques.

6.1 Limitations

Research questions each impose specific requirements on the benchmark that can be used effectively to evaluate them. It is challenging for a single benchmark to satisfy these requirements for a diverse set of research questions, such as the ones we have explored in this paper. For example, the majority of the Defects4J defects have a single failing test, which makes it hard to study the association between the number of failing tests and patch quality. Similarly, a lack of variability in the statement coverage of the developer-written tests makes it hard to study the relationships that involve that coverage. These shortcomings in the benchmark may reduce the strength of the results. Nevertheless, this paper has developed a methodology that can be applied to other benchmarks to further study these questions.

JaRFly, our Java Repair framework, can help future researchers build new Java repair techniques. Our methodology for creating high-quality evaluation test-suites can be used to do so for new benchmarks, and the instances of evaluation test suites we have created for Defects4J can be used for future evaluations on that benchmark in a reproducible manner.

A recent study identified the evaluation-test-suite-based approach to be reproducible, if conservative [72]: Evaluation test-suites may miss identifying some overfitting patches, but every patch they identify as overfitting, does so. This approach is complementary to manual inspection, which is less reliable but can identify some instances of overfitting that evaluation test suites miss [72]. Future research should pursue improving automated test generation with the goal of producing higher-quality evaluation test suites for program repair. Perhaps complementary to this challenge is recent work on automatically generating test-suites from natural-language software artifacts (instead of human-patched version of code) [17], [97].

The generalizability of our results relies on the generalizability of the four program repair techniques we use in our evaluation. While the classification of *G&V* techniques [132] makes the argument that evaluations on representative techniques should generalize to other techniques in this class, evaluations on a larger, more diverse set of techniques provide stronger evidence. In this paper, we have evaluated four *G&V* techniques. Applying our methodology to other techniques would constitute a valuable replication study. However, technological challenges prevented us from adding more techniques. Some projects do not release their tools' implementations, making reuse difficult. Some projects release only compiled binaries of their tools and do not make the source code public, which prevents minor modifications to those tool necessary for running experiments. For example, we were unable to use CapGen [134] in our evaluation because only its compiled binary is publicly available and we could not modify it to run using only a subset of the developer-written test-suites (as is required in Sections 5.2.2 and 5.2.3) and EvoSuite-generated test-suites (as is required in Section 5.2.4). Finally, some tools cannot be used as envisioned by the original project because of environmental changes. For example, we were unable to use ACS [137] in our evaluation because it was designed to work with a particular query style that directly interacts with GitHub, and GitHub has since disabled such queries. More generally, a recent empirical study on Java program repair techniques found that 13 out of the 24 (54 percent) techniques studied could not be used, including ACS and CapGen. The techniques could not be used because they were not publicly available, did not function as expected, required extraordinary manual effort to run (e.g., manual fault localization), or had hard-coded information to work on specific defect benchmarks and could not be modified with reasonable effort to work on others [41]. When possible, future research that produces automated program repair techniques should aim to make their tools public, releasing their source code, and avoid encoding specific benchmarks or experimental set- ups into the tools themselves.

6.2 Threats to Validity

Our study uses Defects4J, a well-established benchmark of defects in five real-world, open source Java projects. The diversity, and real-world nature of Defects4J mitigates the threat that our study will not generalize to other defects. Defects4J is evolving and growing with new projects, and our methodology can be applied to subsequently added projects, and to other benchmarks, to further demonstrate generalizability.

Our objective methodology for measuring patch quality requires independently generated test suites and the quality of those test suites affects our quality measurement. We use state-of-the-art automated test generation techniques, EvoSuite [49] and Randoop [105], but even state-of-the-art tools struggle to perform well on real-world programs. To mitigate this threat, we experimented with two test generation tools and their configuration parameters, developed a methodology for generating and merging multiple test suites, and only perform our study on the 71 out of 106 defects (67 percent) whose evaluation test suites met strict coverage criteria on the code affected by developer-written patches for the defects.

Our test-suite-based methodology for measuring patch quality inherently overestimates the quality of patches because the evaluation test suites are necessarily partial specifications. If our methodology identifies a test that fails on a patch, the patch is necessarily incorrect; however, if our methodology deems a patch of 100 percent quality, there could still exist a hypothetical evaluation test the patch would fail. As a result, our conclusions are conservative. We find that automated program repair often overfits on real-world Java defects, but the reality could be even more dire.

GenProg, Par, SimFix, and TrpAutoRepair are four representative *G&V* automated program repair techniques. Prior work has explored similarity unifying *G&V* repair and developed an underlying theory, suggesting that results from analysis of these four techniques should generalize to other *G&V* techniques [132].

Our methodology follows the guidelines for evaluating randomized algorithms [7] and uses repair techniques' configuration parameters from prior evaluations that explored the effectiveness of those parameter settings [69], [77], [111]. We carefully control for a variety of potential confounding factors in our experiments, and use statistical tests that are appropriate for their context. We make all our code, test suites, and data public to increase researchers being able to replicate our results, explore variations of our experiments, and extend the work to other repair techniques, test suite generation tools, and defect datasets. JaRFly repair framework is available from `http://JaRFly.cs.umass.edu/` and our generated test suites and experimental results from `http://github.com/LASER-UMASS/JavaRepair-replication-package/`.

7 Related Work

This section places our research in the context of prior work on automated program repair (Section 7.1), studies of quality and other properties of automated program repair (Section 7.2), and benchmarks of defects for use to evaluate automated program repair (Section 7.3).

7.1 Automatic Program Repair Techniques

There are two classes of approaches to repairing defects using failing tests to identify faulty behavior and passing tests to encode desirable behavior: *G&V* and semantic-based repair. The *G&V* techniques use search-based software engineering [57] to generate many candidate patches and then validate them against tests. GenProg [77], [80], [133] uses a genetic programming heuristic [71] to search the space of candidate repairs. TrpAutoRepair [111] limits its patches to a single edit, uses random search instead of genetic programming, and heuristics to select which tests to run first, improving efficiency. Prophet [88] and HDRepair [75] automatically learn bug-fixing patterns from prior developer-written patches and use them to produce candidate patches for new defects. AE [132] is a deterministic technique that uses heuristic computation of program equivalence to prune the space of possible repairs, selectively choosing which tests to use to validate intermediate patch candidates. ErrDoc [128] uses insights obtained from a comprehensive study of error handling bugs in real-world C programs to automatically detect, diagnose, and repair the potential error handling bugs in C programs. JAID [26] uses automatically derived state abstractions from regular Java code without requiring any special annotations and employs them, similar to the contract-based techniques to generate candidate repairs for Java programs. Qlose [32] optimizes a program distance, a function of syntactic and semantic differences between the original buggy and the patched programs, while generating candidate patches. DeepFix [56] and ELIXIR [116] use learned models to predict erroneous program locations along with patches. ssFix [135] uses existing code that is syntactically related to the context of a bug to produce patches. CapGen [134] works at the AST node level and uses context and dependency similarity (instead of semantic similarity) between the suspicious code fragment and the candidate code snippets to produce patches. SapFix [89] and Getafix [9], two tools deployed on production code at Facebook, efficiently produce correct repairs for large real-world programs. SapFix [89] uses prioritized repair strategies, including pre-defined fix templates, mutation operators, and bug-triggering change reverting, to produce repairs in real-time. Getafix [9] learns fix patterns from past code changes to suggest repairs for bugs that are found by Infer, Facebook's in-house static analysis tool. SimFix [63] considers the variable name and method name similarity, as well as structural similarity between the suspicious code and candidate code snippets. Similar to CapGen, it prioritizes the candidate modifications by removing the ones that are found less frequently in existing patches. SketchFix [60] optimizes the candidate patch generation and evaluation by translating faulty programs to sketches (partial programs with holes) and lazily initializing the candidates of the sketches while validating them against the test execution. Par [69] and SOFix [84] use predefined repair templates to generate candidate patches. These repair templates are created based on the repair patterns mined from StackOverflow posts by comparing code samples in questions and answers for fine-grained modifications. Synthesis techniques can also construct new features from examples [28], [54], rather than address existing bugs.

The semantic-based techniques use semantic reasoning to synthesize patches to satisfy an inferred specification. Nopol [138], Semfix [101], DirectFix [93], and Angelix [94] use SMT or SAT constraints to encode test-based specifications. S3 [74] extends the semantics-based family to incorporate a set of ranking criteria such as the variation of the execution traces similar to Qlose [32]. JFIX [73] extends Angelix [94] to target Java programs. SemGraft [92] infers specifications by symbolically analyzing a correct reference implementation instead of using test cases. Genesis [85], Refazer [115], NoFAQ [33], Sarfgen [130], and Clara [55] process correct patches to automatically infer code transformations to generate patches. SearchRepair [68] blurs the line between *G&V* and semantic-based techniques by using constraint-based encoding of the desired behavior to replace suspicious code with semantically-similar human-written code from elsewhere.

Our work does not introduce new repair techniques but aims to help techniques properly evaluate their ability to produce high-quality patches for real-world defects. Our work enables properly comparing techniques with respect to patch quality, and encourages the creation of new techniques whose focus is producing high-quality patches on real-world defects. Empirical studies of fixes of real bugs in open-source projects can also improve repair techniques by helping designers select change operators and search strategies [66], [142]. Understanding how repair techniques handle particular classes of errors, such as security vulnerabilities [80], [108] can guide tool design. For this reason, some automated repair techniques focus on a particular defect class, such as buffer overruns [119], [121], unsafe integer use in C programs [29], single-variable atomicity violations [64], deadlock and livelock defects [82], concurrency errors [83], and data input errors [5] while other techniques tackle generic bugs. Our evaluation has focused on tools that fix generic bugs, but our methodology can be applied to focused repair as well.

In addition to repair, search-based software engineering has been used for developing test suites [95], [129], finding safety violations [4], refactoring [117], and project management and effort estimation [11]. 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.

7.2 Empirical Studies Evaluating Automatic Program Repair

Prior work has argued the importance of evaluating the types of defects automated repair techniques can repair [98], and evaluating the generated patches for understandability, correctness, and completeness [96]. Yet many of the prior evaluations of repair techniques have focused on what fraction of a set of defects the technique can produce patches for (e.g., [23], [31], [42], [64], [80], [90], [132], [133]), how quickly they produce patches (e.g., [77], [132]), how maintainable the patches are (e.g., [50]), and how likely developers are to accept them (e.g., [1], [69]).

However, some recent studies have focused on evaluating the quality of repair and developing approaches to mitigate patch overfitting. For example, on 204 Eiffel defects, manual patch inspection showed that AutoFix produced high-quality patches for 51 (25 percent) of the defects, which corresponded to 59 percent of the patches it produced [107]. While AutoFix uses contracts to specify desired behavior, by contrast, the patch quality produced by techniques that use tests has been found to be much lower. Manual inspection of the patches produced by GenProg, TrpAutoRepair (referred to as RSRepair in that paper), and AE on a 105-defect subset of ManyBugs [114], and by GenProg, Nopol, and Kali on a 224-defect subset of Defects4J showed that patch quality is often lacking in automatically produced patches [90]. An automated evaluation approach that uses a second, independent test suite not used to produce the patch to evaluate the quality of the patch similarly showed that GenProg, TrpAutoRepair, and AE all produce patches that overfit to the supplied specification and fail to generalize to the intended specification [20], [122]. This work has led to new techniques that improve the quality of the patches [68], [86], [88], [135], [136], [141]. For example, DiffTGen generates tests that exercise behavior differences between the defective version and a candidate patch, and uses a human oracle to rule out incorrect patches. This approach can filter out 49.4 percent of the overfitting patches [135]. Using heuristics to approximate oracles can generate more tests to filter out 56.3 percent of the overfitting patches [136]. UnsatGuided uses held-out tests to filter out overfitting patches for synthesis-based repair, and is effective for patches that introduce regressions but not for patches that only partially fix defects [141]. Automated test generation techniques that generate test inputs along with oracles [17], [53], [97], [124] or use behavioral domain constraints [6], [52], [65], [127], data constraints [45], [99], [100], or temporal constraints [12], [13], [14], [43], [102] as oracles could potentially address the limitations of the above-described approaches.

Using independent test suites to measure patch quality is imperfect, as test suites are partial and may identify some incorrect patches as correct. On a dataset of 189 patches produced by 8 repair techniques applied to 13 real-world Java projects, independent tests identify fewer than one fifth of the incorrect patches, underestimating the overfitting problem [72]. However, on other benchmarks, the results are much more positive. For example, on the QuixBugs benchmark, combining test-based and manual-inspection-based quality evaluation could identify 33 overfitting patches, while test-based evaluation alone identified 29 of the 33 (87.9 percent) [140]. While the human judgment is a criterion not used by the repair tools for patch construction, it is fundamentally different from the correctness criterion we use in our evaluation, as it is often difficult for humans to spot bugs even when told exactly where to look for them [106]. Further, using independently generated test suites instead of using the subset of the original test suite to evaluate patch quality ensures that we do not ignore regressions a patch is most likely to introduce. Poor-quality test suites result in patches that overfit to those suites [114]. Our evaluation goes further, demonstrating that high-quality, high-coverage test suites still lead to overfitting, and identifying other relationships between test suite properties and patch quality.

Our work has focused on understanding the effectiveness of repair techniques to patch large real-world Java programs correctly and to identify what factors affect the generation of high-quality patches. Studying the effects of test suite size, coverage, number of failing tests, and test provenance on the quality of the patches generated by Angelix on the IntroClass [79] and Codeflaws [126] benchmarks of defects in small programs finds results consistent with ours. By contrast, our work focuses on real-world defects in real-world projects and *G&V* repair. Further, prior work has shown that the selection of test subjects (defects) can introduce evaluation bias [16], [110]. Our evaluation focuses precisely on the limits and potential of repair techniques on a large dataset of defects, and controls for a variety of potential confounds, addressing some of Monperrus' concerns [96].

Our answer to RQ6 considers combining multiple patches in a form of n-version programming [25]. N-version programming works poorly with human-written systems because the errors humans make do not appear to be independent [70]. Our evaluation has shown that the n-version of automatically-generated patches also fails to provide a benefit.

7.3 Defect Benchmarks

Several benchmarks of defects have evolved specifically for evaluating automated repair. The ManyBugs benchmark [79] consists of 185 C defects in real-world software. The IntroClass benchmark [79] consists of 998 C defects in very small, student-written programs, although not all 998 are unique. The Codeflaws benchmark [126] consists of 3,902 defects from 7,436 C programs mined from programming contests and automatically classified across 39 defect classes. The DBGBench benchmark [19] (based on the CoREBench benchmark [18]) contains a collection of 70 real regression errors in four open-source C projects. The QuixBugs benchmark [81] consists of 40 programs from the Quixey Challenge, where programmers were given a short buggy program and one minute to fix the bug. The programs are translated to Python and Java, and each bug is contained on a single line. The Defects4J benchmark [66], originally designed for testing and fault-localization studies, consists of 357 Java defects in real-world software, and has become a popular benchmark for evaluating automated program repair [42], [90], [98], [138]. We elected to use Defects4J because it contains real-world defects in large, complex projects, it supports reproducibility and test suite generation, and is increasingly a testbed for evaluating automated program repair.

8 Contributions

While automated program repair shows promise for improving software quality and reducing the costs of software maintenance, several studies have raised concerns that program repair may do more harm than good in terms of software quality. This paper has systematically and rigorously explored the effect of four *G&V* program repair techniques on real-world defects in real-world Java projects, and found that while program repair techniques do sometimes produce patches, those patches often (between 53.9 and 86.2 percent of the time) break untested or undertested functionality. In fact, the median patch breaks more functionality than it repairs. Increasing the size of the test suite used to guide the repair process can help slightly improve patch quality. In most cases, test suites written by humans lead to higher-quality patches than automatically-generated test suites. Finally, the patches the techniques generate lack sufficient diversity to be combined in a way to improve patch quality.

This work is the first to explore the relationships between these aspects of patch generation and patch quality on real-world defects, building on prior studies on toy programs [20], [76], [122]. Our study rigorously controls for possible confounding factors and uses an objective, repeatable quality-evaluation methodology.

To enable our study, we create JaRFly, a framework for Java *G&V* program repair techniques. We use JaRFly to faithfully reimplement GenProg [77] and TrpAutoRepair [111] for Java, improving on prior attempts to do so. We further use JaRFly to reimplement Par [69] and make the first public release of a Par implementation. JaRFly is open-source and available at `http://JaRFly.cs.umass.edu/`. We further use state-of-the-art automated test generation to generate high-quality test suites for real-world defects in Defects4J used in our study, and create a methodology for generating more such test suites for other defects. Our data, test suites, and scripts are all available at `http://github.com/LASER-UMASS/JavaRepair-replication-package/`.

Overall, our work has identified the shortcomings of today's program repair techniques when applied to real-world defects, and will drive research toward improving the quality of program repair.

Acknowledgments

This work was supported by the US National Science Foundation under grants CCF-1453474, CCF-1563797, CCF-1564162, and CCF-1750116. This work was performed in part using high performance computing equipment obtained under a grant from the Collaborative R&D Fund managed by the Massachusetts Technology Collaborative.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proc. ACM Symp. Operating Syst. Prin.*, 2005, pp. 59–74.

[2] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proc. Annu. Conf. Genetic Evol. Comput.*, 2011, pp. 1427–1434.

[3] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. Le Goues, "SOSRepair: Expressive semantic search for real-world program repair," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2019.2944914.

[4] E. Alba and F. Chicano, "Finding safety errors with ACO," in *Proc. Conf. Genetic Evol. Comput.*, 2007, pp. 1066–1073.

[5] M. Alkhalaf, A. Aydin, and T. Bultan, "Semantic differential repair for input validation and sanitization," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 225–236.

[6] R. Angell, B. Johnson, Y. Brun, and A. Meliou, "Themis: Automatically testing software for discrimination," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2018, pp. 871–875.

[7] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2011, pp. 1–10.

[8] Atlassian, "Clover code coverage tool," 2016. [Online]. Available: https://www.atlassian.com/software/clover

[9] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," in *Proc. ACM Program. Lang. Object-Oriented Program. Syst. Lang. Appl.*, 2019, Art. no. 159.

[10] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2014, pp. 306–317.

[11] A. Barreto, M. Barros, and C. Werner, "Staffing a software project: A constraint satisfaction approach," *Comput. Operations Res.*, vol. 35, no. 10, pp. 3073–3089, 2008.

[12] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Unifying FSM-inference algorithms through declarative specification," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2013, pp. 252–261.

[13] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy, "Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms," *IEEE Trans. Softw. Eng.*, vol. 41, no. 4, pp. 408–428, Apr. 2015.

[14] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, "Leveraging existing instrumentation to automatically infer invariant-constrained models," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2011, pp. 267–277.

[15] A. Bessey *et al.*, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[16] C. Bird *et al.*, "Fair and balanced?: Bias in bug-fix datasets," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2009, pp. 121–130.

[17] A. Blasi *et al.*, "Translating code comments to procedure specifications," in *Proc. Int. Symp. Softw. Testing Anal.*, 2018, pp. 242–253.

[18] M. Böhme and A. Roychoudhury, "CoREBench: Studying complexity of regression errors," in *Proc. ACM/SIGSOFT Int. Symp. Softw. Testing Anal.*, 2014, pp. 105–115.

[19] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? An experiment with practitioners," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 117–128.

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

[21] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.

[22] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: An Eclipse plug-in for testing and debugging," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2012, pp. 378–381.

[23] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2013, pp. 782–791.

[24] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for Web applications," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 237–246.

[25] L. Chen and A. Avižienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Proc. IEEE Int. Symp. Fault-Tolerant Comput.*, 1978, pp. 3–9.

[26] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 637–647.

[27] S. Christou, "Cobertura code coverage tool," 2015. [Online]. Available: https://cobertura.github.io/cobertura/

[28] R. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes, "Program boosting: Program synthesis via crowd-sourcing," in *Proc. Symp. Princ. Program. Lang.*, 2015, pp. 677–688.

[29] Z. Coker and M. Hafiz, "Program transformations to fix C integers," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2013, pp. 792–801.

[30] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (demo)," in *Proc. Int. Symp. Softw. Testing Anal.*, 2016, pp. 449–452.

[31] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 550–554.

[32] L. D'Antoni, R. Samanta, and R. Singh, "QLOSE: Program repair with quantitative objectives," in *Proc. Int. Conf. Comput. Aided Verification*, 2016, pp. 383–401.

[33] L. D'Antoni, R. Singh, and M. Vaughn, "NoFAQ: Synthesizing command repairs from examples," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 582–592.

[34] E. F. de Souza, C. Le Goues, and C. G. Camilo-Junior, "A novel fitness function for automated program repair based on source code checkpoints," in *Proc. Genetic Evol. Comput. Conf.*, 2018, pp. 1443–1450.

[35] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[36] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2010, pp. 65–74.

[37] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, "Inference and enforcement of data structure consistency specifications," in *Proc. Int. Symp. Softw. Testing Anal.*, 2006, pp. 233–243.

[38] B. Demsky and M. C. Rinard, "Goal-directed reasoning for specification-based data structure repair," *IEEE Trans. Softw. Eng.*, vol. 32, no. 12, pp. 931–951, Dec. 2006.

[39] A. Dhar, R. Purandare, M. Dhawan, and S. Rangaswamy, "CLOTHO: Saving programs from malformed strings and incorrect string-handling," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2015, pp. 555–566.

[40] Z. Y. Ding, Y. Lyu, C. Timperley, and C. Le Goues, "Leveraging program invariants to promote population diversity in search-based automatic program repair," in *Proc. Int. Workshop Genetic Improvement*, 2019, pp. 2–9.

[41] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, “Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts,” in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2019, pp. 302–313.

[42] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan, “Automatic repair of real bugs: An experience report on the Defects4J dataset,” *CoRR*, 2015.

[43] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Patterns in property specifications for finite-state verification,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 1999, pp. 411–420.

[44] EclEmma, “JaCoCo Java code coverage library,” 2017. [Online]. Available: https://www.eclemma.org/jacoco/

[45] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.

[46] H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer, “Contracts in practice,” in *Proc. Int. Symp. Formal Methods*, 2014, pp. 230–246.

[47] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, “Designing better fitness functions for automated program repair,” in *Proc. Genetic Evol. Comput. Conf.*, 2010, pp. 965–972.

[48] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, “A genetic programming approach to automated software repair,” in *Proc. Conf. Genetic Evol. Comput.*, 2009, pp. 947–954.

[49] G. Fraser and A. Arcuri, “Whole test suite generation,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.

[50] Z. P. Fry, B. Landau, and W. Weimer, “A human study of patch maintainability,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 177–187.

[51] M. Gabel and Z. Su, “Testing mined specifications,” in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 4.

[52] S. Galhotra, Y. Brun, and A. Meliou, “Fairness testing: Testing software for discrimination,” in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 498–510.

[53] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, “Automatic generation of oracles for exceptional behaviors,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2016, pp. 213–224.

[54] S. Gulwani, “Automating string processing in spreadsheets using input-output examples,” in *Proc. Symp. Princ. Program. Lang.*, 2011, pp. 317–330.

[55] S. Gulwani, I. Radiček, and F. Zuleger, “Automated clustering and program repair for introductory programming assignments,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 465–480.

[56] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, “DeepFix: Fixing common C language errors by deep learning,” in *Proc. Nat. Conf. Artif. Intell.*, 2017, pp. 1345–1351.

[57] M. Harman, “The current state and future of search based software engineering,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2007, pp. 342–357.

[58] M. Harman and B. F. Jones, “Search-based software engineering,” *Inf. Softw. Technol.*, vol. 43, no. 14, pp. 833–839, 2001.

[59] M. R. Hoffmann, B. Janiczak, E. Mandrikov, and M. Friedenhagen, “JaCoCo code coverage tool,” 2009. [Online]. Available: https://www.jacoco.org/jacoco/

[60] J. Hua, M. Zhang, K. Wang, and S. Khurshid, “Towards practical program repair with on-demand candidate generation,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 12–23.

[61] M. Ivanković, G. Petrović, R. Just, and G. Fraser, “Code coverage at Google,” in *Proc. Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 955–963.

[62] J. Jiang, “SimFix implementation,” 2017. [Online]. Available: https://github.com/xgdsmileboy/SimFix/

[63] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, “Shaping program repair space with existing patches and similar code,” in *Proc. ACM/SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 298–309.

[64] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, “Automated atomicity-violation fixing,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2011, pp. 389–400.

[65] B. Johnson, Y. Brun, and A. Meliou, “Causal testing: Understanding defects’ root causes,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2020.

[66] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for Java programs,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.

[67] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 654–665.

[68] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, “Repairing programs with semantic code search,” in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 295–306.

[69] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2013, pp. 802–811.

[70] J. C. Knight and N. G. Leveson, “An experimental evaluation of the assumption of independence in multiversion programming,” *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 1, pp. 96–109, Jan. 1986.

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

[72] D. L. Xuan Bach, L. Bao, D. Lo, X. Xia, S. Li, and C. S. Pasareanu, “On reliability of patch correctness assessment,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2019, pp. 524–535.

[73] D. L. Xuan Bach, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, “JFIX: Semantics-based repair of Java programs via symbolic PathFinder,” in *Proc. ACM/SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 376–379.

[74] D. Le Xuan Bach, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, “S3: Syntax- and semantic-guided repair synthesis via programming by examples,” in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 593–604.

[75] D. L. Xuan Bach, D. Lo, and C. Le Goues, “History driven program repair,” in *Proc. Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 213–224.

[76] D. L. Xuan Bach, F. Thung, D. Lo, and C. L. Goues, “Overfitting in semantics-based automated program repair,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 163–163.

[77] 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 *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2012, pp. 3–13.

[78] C. Le Goues, S. Forrest, and W. Weimer, “Representations and operators for improving evolutionary software repair,” in *Proc. Conf. Genetic Evol. Comput.*, 2012, pp. 959–966.

[79] C. Le Goues *et al.*, “The ManyBugs and IntroClass benchmarks for automated repair of C programs,” *IEEE Trans. Softw. Eng.*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.

[80] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “GenProg: A generic method for automatic software repair,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.

[81] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge,” in *Proc. ACM SIGPLAN Int. Conf. Syst. Program. Lang. Appl.: Softw. Humanity Poster Track*, 2017, pp. 55–56.

[82] Y. Lin and S. S. Kulkarni, “Automatic repair for multi-threaded programs with Deadlock/Livelock using maximum satisfiability,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 237–247.

[83] P. Liu, O. Tripp, and C. Zhang, “Grail: Context-aware fixing of concurrency bugs,” in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 318–329.

[84] X. Liu and H. Zhong, “Mining StackOverflow for program repair,” in *Proc. Int. Conf. Softw. Anal. Evol. Reeng.*, 2018, pp. 118–129.

[85] F. Long, P. Amidon, and M. Rinard, “Automatic inference of code transforms for patch generation,” in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 727–739.

[86] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2015, pp. 166–178.

[87] F. Long and M. Rinard, “An analysis of the search spaces for generate and validate patch generation systems,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2016, pp. 702–713.

[88] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” in *Proc. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2016, pp. 298–312.

[89] A. Marginean *et al.*, “SapFix: Automated end-to-end repair at scale,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2019, pp. 269–278.

[90] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset,” *Empir. Softw. Eng.*, vol. 22, no. 4, pp. 1936–1964, Apr. 2017.

[91] M. Martinez and M. Monperrus, "ASTOR: A program repair library for Java (Demo)," in *Proc. Int. Symp. Softw. Testing Anal. Demo Track*, 2016, pp. 441–444.

[92] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 129–139.

[93] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 448–458.

[94] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 691–701.

[95] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 12, pp. 1085–1110, Dec. 2001.

[96] 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 *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2014, pp. 234–242.

[97] M. Motwani and Y. Brun, "Automatically generating precise oracles from structured natural language specifications," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2019, pp. 188–199.

[98] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" *Empir. Softw. Eng.*, vol. 23, no. 5, pp. 2901–2947, Oct. 2018.

[99] K. Muşlu, Y. Brun, and A. Meliou, "Data debugging with continuous testing," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng. New Ideas Track*, 2013, pp. 631–634.

[100] K. Muşlu, Y. Brun, and A. Meliou, "Preventing data errors with continuous testing," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 373–384.

[101] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2013, pp. 772–781.

[102] T. Ohmann *et al.*, "Behavioral resource-aware model inference," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2014, pp. 19–30.

[103] V. P. L. Oliveira, E. F. de Souza, C. Le Goues, and C. G. Camilo-Junior, "Improved representation and genetic operators for linear genetic programming for automated program repair," *Empir. Softw. Eng.*, vol. 23, no. 5, pp. 2980–3006, 2018.

[104] V. P. L. Oliveira, E. F. D. Souza, C. Le Goues, and C. G. Camilo-Junior, "Improved crossover operators for genetic programming for program repair," in *Proc. Int. Symp. Search Based Softw. Eng.*, 2016, pp. 112–127.

[105] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2007, pp. 75–84.

[106] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 199–209.

[107] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Trans. Softw. Eng.*, vol. 40, no. 5, pp. 427–449, May 2014.

[108] J. H. Perkins *et al.*, "Automatically patching errors in deployed software," in *Proc. ACM Symp. Operating Syst. Princ.*, 2009, pp. 87–102.

[109] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Trans. Evol. Comput.*, vol. 22, no. 3, pp. 415–432, Jun. 2018.

[110] D. Posnett, V. Filkov, and P. Devanbu, "Ecological inference in empirical software engineering," in *Proc. Int. Conf. Automated Softw. Eng.*, 2011, pp. 362–371.

[111] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *Proc. Int. Conf. Softw. Maintenance*, 2013, pp. 180–189.

[112] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 254–265.

[113] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 191–201.

[114] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36.

[115] R. Rolim *et al.*, "Learning syntactic program transformations from examples," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2017, pp. 404–415.

[116] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: Effective object oriented program repair," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 648–659.

[117] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *Proc. Conf. Genetic Evol. Comput.*, 2006, pp. 1909–1916.

[118] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges," in *Proc. Int. Conf. Automated Softw. Eng.*, 2015, pp. 201–211.

[119] S. Sidiroglou and A. D. Keromytis, "Countering network worms through automatic patch generation," *IEEE Security Privacy*, vol. 3, no. 6, pp. 41–49, Nov. 2005.

[120] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2015, pp. 43–54.

[121] A. Smirnov and T. Chiueh, "DIRA: Automatic detection, identification and repair of control-hijacking attacks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2005.

[122] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2015, pp. 532–543.

[123] M. Soto and C. Le Goues , "Using a probabilistic model to predict bug fixes," in *Proc. IEEE 25th Int. Conf. Softw. Anal. Evol. Reeng.*, 2018, pp. 221–231.

[124] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *Proc. Int. Conf. Softw. Testing Verification Validation*, 2012, pp. 260–269.

[125] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 471–482.

[126] S. H. Tan, J. Yi, S. Mechtaev, and A. Roychoudhury, "Codeflaws: A programming competition benchmark for evaluating automated program repair tools," in *Proc. IEEE Int. Conf. Softw. Eng. Poster Track*, 2017, pp. 180–182.

[127] P. S. Thomas, B. C. da Silva, A. G. Barto, S. Giguere, Y. Brun, and E. Brunskill, "Preventing undesirable behavior of intelligent machines," *Science*, vol. 366, no. 6468, pp. 999–1004, Nov. 2019.

[128] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in C," in *Proc. Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2017, pp. 752–762.

[129] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Time-aware test suite prioritization," in *Proc. Int. Symp. Softw. Testing Anal.*, 2006, pp. 1–12.

[130] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 481–495.

[131] Y. Wei *et al.*, "Automated fixing of programs with contracts," in *Proc. Int. Symp. Softw. Testing Anal.*, 2010, pp. 61–72.

[132] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 356–366.

[133] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2009, pp. 364–374.

[134] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 1–11.

[135] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 226–236.

[136] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2018, pp. 789–799.

[137] Y. Xiong *et al.*, "Precise condition synthesis for program repair," in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2017, pp. 416–426.

[138] J. Xuan *et al.*, “Nopol: Automatic repair of conditional statement bugs in Java programs,” *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.
[139] J. Xuan and M. Monperrus, “Test case purification for improving fault localization,” in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 52–63.
[140] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, “A comprehensive study of automatic program repair on the QuixBugs benchmark,” in *Proc. IEEE Int. Workshop Intell. Bug Fixing*, 2019, pp. 1–10.
[141] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, “Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system,” *Empir. Softw. Eng.*, vol. 24, no. 1, pp. 33–67, Feb. 2019.
[142] H. Zhong and Z. Su, “An empirical study on real bug fixes,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2015, pp. 913–923.

Manish Motwani received the MS degree from the University of Massachusetts Amherst, Amherst, Massachusetts, in 2018. He is currently working toward the PhD degree in the College of Information and Computer Sciences, University of Massachusetts Amherst, Amherst, Massachusetts. His research involves studying large software repositories to learn interesting phenomena in software development and maintenance, and to use that knowledge to design novel automation techniques for testing and program repair. For more information, please visit http://people.cs.umass.edu/~mmotwani/.

Mauricio Soto received the MS degree from the Carnegie Mellon University, Pittsburgh, Pennsylvania, in 2018. He is currently working toward the PhD degree in the School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. His research focuses on improving automated program repair techniques. For more information, please visit https://www.cs.cmu.edu/~msotogon/.

Yuriy Brun (Senior Member, IEEE) received the PhD degree from the University of Southern California, Los Angeles, California, in 2008 and completed his postdoctoral work with the University of Washington, in 2012. He is an associate professor with the College of Information and Computer Sciences, University of Massachusetts Amherst. His research focuses on software fairness, testing, and analysis. He received an NSF CAREER Award, an IEEE TCSC Young Achiever in Scalable Computing Award, and the SEAMS 2020 Most Influential Paper Award. He is a distinguished member of the ACM. For more information, please visit http://www.cs.umass.edu/~brun/.

René Just is an assistant professor with the University of Washington. His research interests include software engineering and software security, in particular static and dynamic program analysis, mobile security, mining software repositories, and applied machine learning. His research in the area of software engineering won three ACM SIGSOFT distinguished paper awards, and he develops research infrastructures and tools (e.g., Defects4J and the Major mutation framework) that are widely used by other researchers. For more information, please visit https://homes.cs.washington.edu/~rjust/.

Claire Le Goues (Member, IEEE) received the BA degree in computer science from Harvard University, Cambridge, Massachusetts, and the MS and PhD degrees from the University of Virginia, Charlottesville, Virginia. She is an associate professor with the School of Computer Science, Carnegie Mellon University, where she is primarily affiliated with the Institute for Software Research. She received an NSF CAREER Award, the ICSE 2019 Most Influential Paper Award, and the ACM SIGEVO Impact Award in 2019. She is interested in constructing high-quality systems in the face of continuous software evolution, with a particular interest in automatic error repair. For more information, please visit http://www.cs.cmu.edu/~clegoues.