The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs

Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, *Member, IEEE*, Premkumar Devanbu, Stephanie Forrest, *Fellow, IEEE*, Westley Weimer

Abstract—The field of automated software repair lacks a set of common benchmark problems. Although benchmark sets are used widely throughout computer science, existing benchmarks are not easily adapted to the problem of automatic defect repair, which has several special requirements. Most important of these is the need for benchmark programs with reproducible, important defects and a deterministic method for assessing if those defects have been repaired. This article details the need for a new set of benchmarks, outlines requirements, and then presents two datasets, MANYBUGS and INTROCLASS, consisting between them of 1,183 defects in 15 C programs. Each dataset is designed to support the comparative evaluation of automatic repair algorithms asking a variety of experimental questions. The datasets have empirically defined guarantees of reproducibility and benchmark quality, and each study object is categorized to facilitate qualitative evaluation and comparisons by category of bug or program. The article presents baseline experimental results on both datasets for three existing repair methods, GenProg, AE, and TrpAutoRepair, to reduce the burden on researchers who adopt these datasets for their own comparative evaluations.

Index Terms—Automated program repair, benchmark, subject defect, reproducibility, MANYBUGS, INTROCLASS.

1 INTRODUCTION

REPRODUCIBLE research is a concern throughout science. In a 2013 article "How Science Goes Wrong", *The Economist* criticized research validity in fields ranging from biotechnology ("half of published research cannot be replicated") to computer science ("three-quarters of papers in [a] subfield are bunk") [29]. Similarly, in early 2014, the President's Commission to Advance Science and Technology (PCAST) held a meeting devoted to the many problems of irreproducible and incomparable results in science [64]. The carefully controlled, reproducible experiment is a bedrock principle of modern science, but as these two examples highlight, it is a concept that is more easily stated than implemented. Problems arise from poor statistical methodology, sloppy experimental design, inadequate reviewing, and idiosyncratic data sets. Computer science has historically addressed this last problem through the use of standardized benchmark problems, e.g., [10], [13], [72]. A well-designed benchmark set simplifies experimental reproduction, helps ensure generality of results, allows direct comparisons between competing methods, and over time enables measurement of a field's technical progress.

A common set of benchmarks and evaluation methodologies are good for a research subfield in additional ways. They stimulate cohesion, collaboration, and technical progress within a community. Sim *et al.* argue that benchmarks capture a discipline's dominant research paradigms and represent (and by extension, can promote) consensus on which problems are worthy of study [70]. When a research subfield reaches a sufficient level of maturity, common sets of study objects and baselines, i.e., benchmarks, become instrumental for further progress.

Since 2009, research in automated program repair, a subfield of software engineering, has grown to the point that it would benefit from carefully constructed benchmarks. Software quality in general, and software maintenance in particular, remain pressing problems [16]. Defect repair is critical to software maintenance, and there has been significant progress in automated approaches to repairing software defects. Research in automated repair has grown considerably in the past decade, as evidenced by the number of separate projects in the research literature today (e.g., [20], [22], [25], [44], [47], [52], [58], [63], [73], [76]). This influx of new ideas is exciting and suggests a promising future for automated software engineering in research and practice. Few of these publications, however, include direct empirical comparisons with other approaches. Thus, it is currently difficult to evaluate how different algorithms or assumptions perform relative to one another, or to particular classes of defects or programs.

We believe that the rigorous, reproducible evaluation of and between techniques is critical to allowing researchers to move from questions such as "can this be done at all?" to "why does this work, and under what circumstances?" For example, since 2002, the "MiniSAT Hack" standardized

- *C. Le Goues is with the School of Computer Science at Carnegie Mellon University, Pittsburgh, PA 15213. Email: clegoues@cs.cmu.edu*
- *N. Holtschulte and S. Forrest are with the Department of Computer Science at the University of New Mexico, Albuquerque, NM 87131. Email: {neal.holts, forrest}@cs.unm.edu*
- *E.K. Smith and Y. Brun are with the College of Information and Computer Science at the University of Massachusetts at Amherst, Massachusetts, 01003. Email: {tedks, brun}@cs.umass.edu*
- *P. Devanbu is with the Department of Computer Science at the University of California at Davis, 95616. Email: devanbu@cs.ucdavis.edu*
- *W. Weimer is with the Department of Computer Science at the University of Virginia, Charlottesville, VA 22904. Email: weimer@cs.virginia.edu*

benchmarks and challenges (associated with the International Conferences on the Theory and Applications of Satisfiability Testing, e.g., SAT 2013 [40] and SAT 2014 [30]) have helped foster significant improvement and interest in state-of-the-art SAT solvers. While we are not proposing an explicit human competition in the vein of the MiniSAT Hack, experiments that improve on prior results using the same methodology and benchmarks do lead to direct competition and the ability to compare and contrast research advances. This sort of direct competition is possible only when benchmarks are available to serve as a basis for comparison.

Popular benchmarks such as SPEC [72] are inappropriate for automated repair research because they do not contain explicit defects, and the common practice of manually selecting evaluation subjects for each new technique, while reasonable in a nascent subfield, cannot ultimately establish general results. This suggests a need for new benchmarks, tailored to the requirements of research in automated program repair.

This article presents two benchmark sets consisting of defects[1] in C programs: ManyBugs and IntroClass. The benchmarks are available for download: **http://repairbenchmarks.cs.umass.edu/**. Both benchmark sets are designed to support research on automatic program repair, targeting large-scale production programs (ManyBugs) and smaller programs written by novices (IntroClass). We describe the methodologies in detail to encourage the community to provide additional scenarios and keep the benchmarks up to date as new types of programs and bugs emerge. We also provide baseline results of existing techniques on these benchmarks, against which future research can be compared.

The primary contributions of this article are:

- The ManyBugs dataset, which consists of 185 defects in nine open-source programs. The programs are large, popular, open-source projects, and each defect has at least one corresponding patch and test case written by an original developer. The defects were captured systematically through version control repositories (rather than from explicit developer annotations) from all viable version control versions within given time windows. The goal is to support indicative longitudinal studies within each program and latitudinal studies across programs. To ensure reproducibility, we provide virtual machine images on which the programs and defects are known to execute as described. Each program and defect pair has been manually categorized and classified, providing a starting place for qualitative analyses of techniques. In total, the ManyBugs benchmark programs include 5.9 million lines of code and over 10,000 test cases.
- The IntroClass dataset, which consists of 998 defects in student-written versions of six small C programming assignments in an introductory, undergraduate C programming course. Each of the six assignments asks students to write a C program that satisfies a carefully written specification and allows them to test under-development programs against an instructor-written test suite. The defects (test case failures) are collected from these tested versions. For each such defective program, the benchmark also includes the final program submitted by that student, some of which pass all of the test cases. The six oracle programs, each implementing one of the six assignment specifications, total 114 lines of code. Each assignment's two independent test suites comprise 95 tests in total. The programs' small sizes, well-defined requirements, and numerous varied human implementations enable studies that would be difficult with ManyBugs, particularly studies benefiting from larger numbers of defects, or of techniques that do not yet scale to real-world software systems or involve novice programmers. IntroClass also supports comprehensive, controlled evaluations of factors that affect automated program repair.
- A qualitative analysis and categorization of the benchmark programs and defects. We manually categorize the bugs by defect feature, providing a starting point to evaluate strengths and weaknesses of a given repair technique with respect to given defect types.
- Baseline experimental results and timing measurements for three existing repair algorithms, GenProg [51], [52], [76], AE [75], and TrpAutoRepair [66].

An initial 105-defect subset of the ManyBugs benchmark set was previously used as part of a systematic evaluation of one repair method, GenProg [49]. A subset of those original scenarios was similarly used by others to evaluate TrpAutoRepair [66] (TrpAutoRepair was also published under the name RSRepair in "The strength of random search on automated program repair" by Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang in the 2014 International Conference on Software Engineering; in this article, we use the original name, as it is associated with the complete algorithm description.) The IntroClass benchmark was similarly used in another evaluation of GenProg [17], [71]. This article focuses on the benchmarks and associated generation methodology. The ManyBugs benchmark significantly extends the previous set, and includes an additional subject program (`valgrind`) and additional scenarios from other subjects (`python`, `php`, `wireshark`). The expansion of the `php` set enables an additional type of longitudinal experiment of repeated defect repair in one subject program. We have also modified and added more rigorous validation of all of the scenarios to provide stronger guarantees about their utility in automatic program repair experiments. Finally, we improved the usability of the IntroClass benchmark, categorized all of the defects, and formalized the methodology used to create the dataset.

The rest of this article proceeds as follows. Section 2 discusses the importance and increasing prevalence of research in automatic program repair and the problems with current empirical evaluations in the area. Section 3

1. We use the terms *bug* and *defect* interchangeably throughout this article, referring to the colloquial meaning, that is, undesired program behavior that a programmer would like to repair.

outlines our goals in designing and evaluating the benchmark sets. Sections 4 and 5, respectively, present the methodology for generating the MANYBUGS and INTROCLASS benchmarks, outline the programs and defects, and overview the released scenarios. Section 6 presents baseline empirical repair results for three automated repair methods, GenProg, TrpAutoRepair, and AE, on the two benchmark sets. Section 7 places our work in the context of related research, Section 8 outlines threats to validity, and Section 9 summarizes our contributions and conclusions.

2 MOTIVATION

We motivate the need for widely accessible benchmarks for research in automated program repair by first emphasizing the importance of the research area, then discussing the increasing maturity of the subfield, and finally outlining some deficiencies in current empirical evaluations.

2.1 Program repair is an important problem

The cost of debugging and maintaining software has continued to rise, even while hardware and many software costs fall. In 2006, one Mozilla developer noted, "everyday, almost 300 bugs appear [. . .] far too much for only the Mozilla programmers to handle" [6, p. 363]. The situation has hardly improved in the intervening years, as bugzilla.mozilla.org indicates similar rates of bugs reported in 2013. A 2013 study estimated the global cost of debugging at $312 billion, with software developers spending half their time debugging [16]. Since there are not enough developer resources to repair all of these defects before deployment, it is well known that programs ship with both known and unknown bugs [53].

In response to this problem, many companies offer *bug bounties* that pay outside developers for candidate repairs to their open source code. Well-known companies such as Mozilla ($3,000/bug) [1], Google ($500/bug) [2], and Microsoft ($10,000/bug) [3], ($10,000/bug) offer significant rewards for security fixes, reaching thousands of dollars and engaging in bidding wars [79]. While many bug bounties simply ask for defect reports, other companies, such as Microsoft, reward defensive ideas and patches as well (up to $50,000/fix) [4].

The abundance and success of these programs suggests that the need for repairs is so pressing that some companies must consider outside, untrusted sources, even though such reports must be manually reviewed, most are rejected, and most accepted repairs are for low-priority bugs [5]. A technique for automatically generating patches, even if those patches require human evaluation before deployment, could fit well into this paradigm, with potential to greatly reduce the development time and costs of software debugging. These examples also suggest that benchmarks for automatic program repair should address success metrics relevant to real-world debugging, which include the fraction of queries that produce code patches, monetary cost, and wall-clock time cost.

2.2 Automatic program repair

The importance of defects in software engineering practice is reflected in software engineering research. Since 2009, when automated program repair was demonstrated on real-world problems (PACHIKA [23], ClearView [63], GenProg [76]), interest in the field has grown steadily, with multiple novel techniques proposed (e.g., Debroy and Wong [25], AutoFix-E [73], ARMOR [21], [20], AFix [44], AE [75], Coker and Hafiz [22], PAR [47], SemFix [58], TrpAutoRepair [66] Monperrus [56], Gopinath *et al.* [35], MintHint [46], etc.). Some of these methods produce multiple candidate repairs, and then validate them using test cases, such as by using stochastic search or methods based on search-based software engineering [37] (e.g., GenProg, PAR, AutoFix-E, ClearView, Debroy and Wong, TrpAutoRepair). Others use techniques such as synthesis or constraint solving to produce smaller numbers of patches that are correct by construction (e.g., Gopinath *et al.*, AFix, etc.) relative to inferred or human-provided contracts or specifications. We provide a more thorough treatment of related work in Section 7.

Several recent studies have established the potential of these techniques to reduce costs and improve software quality, while raising new questions about the acceptability of automatically generated patches to humans. See, for example, the systematic study of GenProg, which measured cost in actual dollars [49] and related studies that assess the acceptability of automatically generated patches [47], [33].

As automatic repair research has matured, interest has moved from proof-of-concept evaluations to broader qualitative questions that identify the circumstances under which automatic repair is most appropriate. As an indicative example of conventional wisdom, Thomas Zimmermann of Microsoft Research claimed that, "one of the challenges will be to identify the situations when and where automated program repair can be applied. I don't expect that program repair will work for every bug in the universe (otherwise thousands of developers will become unemployed), but if we can identify the areas where it works in advance there is lots of potential." [74, slide 67][2]

Taken together, this body of work and commentary show the promise of automated software repair methods, and it highlights the need to understand and improve the quality of the automated repair methods before they can be adopted widely in practice. The rapid growth and increasing maturity of the research area suggests a need for community discussion and consensus on evaluation methodology. While the work to date is promising in providing proofs of concept, larger-scale, generalizable and comparative evaluations will be necessary to support conclusions about practical feasibility.

2.3 Empirical evaluations

The vast majority of published papers on software defect detection, localization, and repair use empirical evaluation

2. http://www.cs.virginia.edu/~weimer/p/weimer-ssbse2013.pdf

to demonstrate the effectiveness of the method. Typically, a repair technique is proposed and then tested on a collection of bugs in one or more programs. When chosen carefully, such studies provide evidence that the technique in question will generalize or scale to different or larger projects. Studies of this type tend to use bugs from one or a combination of three sources:

- **Defects in the Wild.** A number of studies of automatic bug finding, localization, and fixing techniques have used bugs "in the wild," found through ad hoc case studies, manual search through bug databases, industrial partnerships, and word-of-mouth (e.g., [53], [63], [52]). We previously took this approach to evaluate GenProg on a variety of defect types [52]. Similarly, ClearView [63] was evaluated (both in terms of patching success and via a Red Team evaluation) on a set of previously identified vulnerabilities from historical bug reports for the FireFox web browser; these bugs fell into particular defect classes that are handled by ClearView's underlying vulnerability detector.
- **Systematic search.** A dataset constructed by collecting defects in a systematic fashion helps avoid biased evaluations. The benchmark set (described earlier) that was created to support a large-scale evaluation of GenProg was one of the first examples of a publicly available, systematically assembled set of bugs constructed specifically for studies of automated program repair [49]. The approach behind the iBugs [24] and Defects4J [45] share some features with the ManyBugs methodology but focus on Java benchmarks suitable for evaluating research in fault localization and testing, respectively. These datasets or the underlying methodologies could possibly be adapted for systematic evaluations of automatic program repair research, although we are unaware of any such efforts to date.
- **Existing datasets.** There are several existing repositories of programs associated with test suites, some of which can be and have been adapted for empirical evaluations of program repair research. These repositories were designed for research on software testing (similar to iBugs). For example, the Software Artifact Infrastructure Repository (SIR) [27], is described by its maintainers as "a repository of software-related artifacts meant to support rigorous controlled experimentation with program analysis and software testing techniques, and education in controlled experimentation." Some evaluations have used the SIR repository, focusing primarily on the subset constituting the Siemens benchmark suite [39], which currently consists of 8 programs of 173–726 lines of code. The SIR C suite overall contains 15 programs ranging from 173 lines (`tcas`) to over 120,000 lines (`vim`), although we are unaware of evaluations that include the latter. Because SIR is intended to support controlled evaluations of testing research, the test suites were created to maximize coverage, and the majority of the faults are seeded. These conditions are not representative of real-world development, where, for example, test suite coverage is often far from maximal and faults occur naturally during development. The SAMATE benchmark [57], constructed by NIST to demonstrate particular common classes of security errors in design, source code, and binaries. Coker and Hafiz used some of these programs in an empirical evaluation [22]. The Defects4J benchmark [45] includes 357 bugs and 20,109 tests in 5 real-world programs, and targets testing techniques, specifically focusing on mutation testing. Evaluations have used Defects4J to show that mutants are a valid substitute for real faults in software testing [45] and to measure how likely humans are to accept automatically generated patches [28]. These datasets are most suitable for the controlled experiments for which they were designed, such as measuring success relative to test suite coverage or defect type.

Few of the published automated software repair projects report direct comparisons to previous work, which motivates our inclusion of baseline repair results. However, three recent publications do include empirical comparisons, each featuring a different repair method: PAR [47], SemFix [58], and TrpAutoRepair [66]. All three compare new and existing techniques' performance on a new dataset (SemFix and TrpAutoRepair compared to previously released versions of GenProg, and PAR compared to a GenProg reimplementation for Java). Such approaches are scientifically valuable because they make direct empirical comparisons, and reuse and reproduce an existing technique. However, repeatedly selecting new benchmarks and reimplementing algorithms duplicates work, and introduces subtle variations to technique implementations.

Direct comparisons are rare in part because it is often difficult to reproduce both the defects to be repaired and the existing techniques. Although publicly available codebases help the latter problem, exact replication of experimental conditions typically requires significantly more detailed information than can be included in a standard conference publication. Reproducing a defect requires knowledge of, and access to, at least (1) the exact version of the source code containing the defect; (2) the test case(s) or specifications used to demonstrate the problem and specify correct behavior, if applicable; (3) instructions for running test cases for a particular tool, e.g., as part of a framework (JUnit) or individually, perhaps with a custom script; (4) compilation mechanism, including the specific compiler and its version, or other support scripts; and (5) the operating system or platform and all associated libraries, which can impact the defect or how it manifests, e.g., integer overflows on 32- vs 64-bit architectures.

A second set of difficulties arises if the repair methods have stochastic elements and a large set of parameters. In many cases, these are not delineated in enough detail to replicate results. Although researchers are increasingly releasing specific details in technical reports or other external data artifacts to support reproducibility, such efforts are not yet the norm.

One critical concern in empirical evaluation is how the dataset is selected. For automated repair, for example, we are interested in how defects are sampled from the real world. A recent critical review of GenProg and PAR discusses this issue in detail [56, Sec. 3.2.1]: A benchmark made entirely of bugs of category X favors techniques that perform well on X. Ideally, a benchmark set for this domain will be indicative of bugs encountered in development practice, and defect categories will occur in the benchmark set in the same proportion that they occur in reality. In practice, such determinations are hard to make, especially in the face of ever-changing technology. This is evident in the evolution of the popular SPEC benchmarks, which are ubiquitous in evaluating compiler optimizations [72], and which are often reselected and revised as the field and available hardware advances.

Motivated by these issues, this article presents a benchmark set of program defects that are deterministic and reproducible, with the goal of enhancing the a wide range of experiments with different methods for automatic program repair.

3 Generating the benchmarks

A *benchmark* is a standard or point of reference against which something can be compared or assessed. A benchmark for evaluating program repair techniques consists of a program with a defect and a mechanism for evaluating the correctness of a repair for that defect. A benchmark set is a set of such defects in programs. In the ideal case, the benchmark set allows quantitative and qualitative analysis of the characteristics of a given program repair technique. Complete evaluation on a particular defect requires a *defect scenario*, which consists of: a software system with a defect; the environment it runs in; a witness to the defect; a specification of correct behavior; and optionally, a human-supplied repair. This section describes these requirements for both complete defect scenarios and a collection of them.

3.1 Individual scenarios

A *defect scenario* consists of:

- A partial or complete *software system* with a defect (for example, a source-tree snapshot of an off-the-shelf C program, including Makefiles). This is the program that a repair algorithm will attempt to repair. Repair methods target different representation levels, including the source code text itself, the compiled binary, or an intermediate representation such as an abstract syntax tree (AST) or LLVM Intermediate Representation. Including the full source tree and compilation mechanisms allows the benchmarks to support experiments at any of these representation levels.
- An *environment* in which the program can be reliably and deterministically compiled and executed. Many repair techniques are dynamic or have dynamic components, and thus require the ability to compile and execute both the original program and variants. The size and complexity of the software system under repair typically contributes to the complexity of the environment. Reproducing a set of program behaviors deterministically often depends on factors like system architecture, installed libraries, or operating system. Thus, for the larger programs in the ManyBugs set, we provide a virtual machine image that includes all necessary library versions and compilation scripts (see Section 4.4). This simplifies reproduction of the defects in question. By contrast, and by design, the smaller programs in the IntroClass set have few environmental dependencies, and thus can be compiled and run reliably given a set of common flags to the `gcc` compiler.
- A *witness* to the defect, such as one or more test cases that the program fails. This identifies the bug, usually corresponding to a test case the software system is currently failing. The witness also identifies repairs for the defect by describing expected correct behavior.
- A (partial) *specification* of correct behavior. For example, a test suite produced by the original application developers that the program snapshot currently passes. Although test cases are at best partial and imperfect measures of program correctness or quality, they are by far the most prevalent technique for establishing correctness in practice. Other specification mechanisms, such as invariants, annotations, or supportive documentation, could also fulfill this requirement as available.
- Optionally, an example human-written repair, or *fix*. For example, if the defect in question was repaired in the program's source code repository, the scenario may include the patch from that repository. Such patches assist in qualitative and quantitative post facto evaluations by providing a point of comparison.

As mentioned earlier, deterministically reproducing program behavior (both correct and incorrect) on a set of test cases can be challenging, and our benchmarks thus focus on deterministic defects. Race conditions and other nondeterministic defects [44] are certainly important, but they are outside the scope of this benchmarking effort and entail a different set of concerns with respect to evaluation [15]. Limiting the scope to deterministic faults supports our ability to provide the following guarantees for each scenario:

- The code compiles correctly in the environment provided with the commands provided.
- The compiled buggy code passes all elements of the specification, or all normal test cases, *except* those that witness the defect.
- The compiled code demonstrates the defect (i.e., fails all tests that encode the bug).
- Example repairs included in the benchmark produce programs that compile and pass all elements of the specification and fail to witness the defect (i.e., retain all required functionality while repairing the bug).
- Claims regarding deterministic behavior, e.g., that the program passes all of the positive test cases, is

validated with respect to a combination of manual inspection (where applicable) and at least 100 trial executions, all of which were observed to produce the identical result. Any remaining exceptions (limited to the IntroClass set, see below for details) are noted carefully in the released data. All tests that involve timing (e.g., tests that include time outs to guard against infinite loops) were observed to complete in time over 100 trials on 2013 Amazon EC2 cloud computing c1.medium instances.

- No part of the specification (e.g., no test case) can be satisfied by a program with explicitly degenerate behavior, such as a program that crashes immediately on launch.
- The order in which two specification elements or two candidate programs are checked (e.g., in which order two test cases are run or two candidate repairs are tested) does not influence the result. That is, evaluations are *stateless*, even if the programs are ill-behaved.

Some of these guarantees merit further explanation. The specification and witness test cases provide a way to run a sanity check on a scenario before beginning a repair effort. It is important to establish that the original program compiles, demonstrates the defect, and otherwise works correctly. Some human-written test cases are nondeterministic (e.g., involve random numbers or machine-specific timeouts). Because our benchmarks assume determinism, we removed such test cases from consideration. However, one interesting source of nondeterminism occurred in the IntroClass benchmark set, where in some cases, student programmers used uninitialized variables or out-of-bounds array reads. We verified that in our execution environment, each of the program versions in IntroClass produced consistent results over at least 10 executions.

A full evaluation of the quality, maintainability [33] or acceptability [47] of a repair is beyond the scope of this work. Test cases do not provide the final word on repair quality, and future work is needed to develop methods for measuring repair quality beyond test suite behavior. In the absence of such metrics, however, the final three properties listed above provide additional confidence in the quality of a machine-generated repair. They address corner cases that we have observed in practice, where repairs that cause a program to pass all test cases are unlikely to be acceptable to humans. As one example (explicitly degenerate behavior), many test suites are written assuming cooperating developers, and do not rule out truly pathological behavior. For example, a common test practice runs the program and then searches the output for a known bad substring; if the substring is absent, the test passes. This type of test can be trivially passed by an empty program that returns immediately, which is rarely the truly desired behavior. Further, many larger programs use bootstrapping in compilation or testing (e.g., first automatically generating header files). Unless care is taken to restore a pristine environment between test evaluations [80], a low-quality candidate repair (e.g., one that creates malformed headers during bootstrapping or that fails to properly cleanup after a test case) may cause all subsequent evaluated candidates to appear to fail, violating the *stateless* requirement.

Although they do not provide formal guarantees, these three properties add confidence, especially compared to previous evaluations (e.g., [49]), that candidate repairs that pass all specification elements are not degenerate. With interest in automated program repair methods on the rise, there is a need for improved methods for partial correctness specifications that are robust to automatically generated code. To date, we have not found other types of these specifications in the types of open-source programs we investigated for these benchmarks.

3.2 The sets of scenarios

We now describe the set of experimental questions and analyses that are common in current automatic program repair research. We used these to determine requirements for our benchmark sets with the goal of enabling multiple classes of experiments:

1) Program Generality. Does the repair technique apply to multiple classes of software programs? Or does it target or work best on one class of applications (e.g., web servers)?
2) Defect Generality. Does the repair technique apply to multiple classes of software defects, or does it successfully target one class of bugs (e.g., buffer overflows)?
3) Longitudinal Studies. Can the repair technique fix multiple defects in the same program? Does the repair technique apply to iterative repair scenarios in which additional defects are reported over time?
4) Success Characteristics. What characterizes defect scenarios on which the technique succeeds (e.g., high priority defects, defects of a certain size, mature host programs, programs of a particular size, etc.)?
5) Controlled Evaluations. How does a technique perform on hundreds of defects? How do factors like test suite coverage influence performance?
6) Human Comparisons and Overall Repair Quality. How do the repairs produced by the technique compare to repairs produced by humans (e.g., students or developers)? Are the repairs produced by such techniques of acceptable quality?

There are several plausible use cases addressed by automated program repair. For example, some repair techniques may be designed to address particular defect or program types; others to be generally applicable; still others to address iterative, closed-loop modification of long-running systems. Our benchmarks support evaluation of a variety of use cases and both latitudinal studies (many types of programs and defects) and longitudinal studies (many defects in the same program over time). To admit principled comparisons and identify fruitful areas of future research, our benchmarks are *categorized* (e.g., "technique A performs well on defects of type B not on those of type C").

It is desirable to include scenarios that are *indicative* of the defects encountered by developers in the real-world. These defects ideally correspond to high-priority bugs in commonly used programs. An important aspect of this property is that the defects should be *systematically identified*, avoiding cherry-picking of ideal candidates. The closer such a defect set is to those that appear in practice, the more results on them will be indicative and generalizable to real-world applications. This provides another reason to include developer-provided test cases as the partial specifications of both correct and incorrect behavior: these test cases are indicative of the types of evidence available in practice for evaluating program functionality.

In addition, automatic repair techniques vary in their maturity. One dimension along which techniques vary is scalability. Ideally, a benchmark enables characterization of a technique's scaling behavior. We thus include programs of varying size, some of which will by necessity be more indicative of real world code than others.

Finally, the scenarios must support *controlled* and *reproducible* experiments. We addressed some reproducibility issues through careful definition of defect scenario (Section 3.1); many of the defects require particular library versions or otherwise cannot easily be reproduced simply from a program, a source control repository, and a revision number specification. This motivated us to include virtual machine images to support experimentation on the larger MANYBUGS programs. However, the issue of controlled experimentation applies at the benchmark level as well. Ideally, the scenarios are selected to allow a researcher to control for various features of a defect or program ahead of time as well as in post facto analysis.

3.3 Baseline repair data

Many benchmarks are associated with baseline measurements against which future evaluations can be compared (e.g., the SPEC benchmarks). We expect future researchers to expand the dimensions along which automatic repair techniques are evaluated beyond time, efficiency, or cost. However, as the state-of-the-art in industrial practice motivates the use of these metrics as one way of evaluating success (see Section 2.1), we use them as baseline metrics. Baselines reduce the experimental burden of comparative evaluations on other researchers, so long as the future evaluations use the scenarios as-is (for example, modifying the test suites would necessitate a full re-execution of the baseline evaluation). Finally, demonstrated integration of existing repair methods with the defect scenarios in this benchmark provides a concrete model for how other tools can be evaluated using these benchmarks.

For each scenario, we report if GenProg (version 2.2, with its latest published parameters [50]), AE [75], and TrpAutoRepair [66] produce a repair, and how long such a search took, using the parameters and settings from its most recently-published methodology. GenProg and TrpAutoRepair were allotted 10 random seeds of up to 12 hours each (totaling 120 hours). AE was allotted one deterministic trial of up to 60 hours, as per its published algorithm description. Because the INTROCLASS scenarios take substantially less time to process, we ran GenProg and TrpAutoRepair on 20 random seeds instead of 10, which may lead to stronger statistical results [8]. Although not indicative of all program repair approaches, GenProg, AE and TrpAutoRepair represent a family of generate-and-validate repair architectures based around testing, and GenProg has been used as a comparison baseline in several evaluations (e.g., [47], [58], [67], [66]).

4 THE MANYBUGS BENCHMARK

The MANYBUGS benchmark consists of 185 defect scenarios, constructed according to the requirements described in Section 3.1. The benchmark is designed to allow indicative evaluations whose results generalize to industrial bug-fixing practice, while allowing qualitative discussions of the types of programs and defects on which a particular technique is successful. Because we generate the scenarios systematically over the history of real-world programs, the set is less suitable for *controlled* experimentation, in which factors like test suite size or initial program quality are varied in a principled way. The INTROCLASS benchmark, described in Section 5, is intended to support those types of experiments. It is also comprised of small programs, rendering it more suitable for evaluating novel methods that may not immediately scale to large legacy programs.

This section discusses the methodology we used to construct the MANYBUGS benchmark (Section 4.1), providing details about each of the benchmark programs (Section 4.2), characterizing the defect scenarios across several dimensions (Section 4.3), and presenting a high-level outline of the individual scenarios (Section 4.4).

The MANYBUGS benchmark may evolve. This section and the results in Section 6 describe MANYBUGS v1.0.

4.1 Methodology

Our goal in constructing the MANYBUGS benchmark was to produce an unbiased set of programs and defects that is indicative of "real-world usage."[3] We therefore sought *subject programs* that contained sufficient C source code, and included a version control system, a test suite of reasonable size, and a set of suitable subject defects. We focused on C because, despite its age, it continues to be the most popular programming language in the world [65], and because a large proportion of the existing research projects in automatic repair focus on bugs in C programs. For the purposes of reproducibility, we adopted only programs that could run without modification in a common denominator cloud computing virtualization (see

3. Some of the material in this section was previously presented in [49]. We have adapted and contextualized the text for use here; some of it remains unchanged.

Section 3.1). This limited us to programs amenable to such environments.

At a high level, to identify a set of *subject defects* that were both reproducible and important, we searched systematically through the program's source history, looking for revisions at which program behavior on the test suite changes. Such a scenario corresponds either to a human-written repair for the bug corresponding to the failing test case, or a human-committed regression. This approach succeeds even in projects without explicit bug-test links (which can lead to bias in bug datasets [11]), and ensures that benchmark defects are sufficiently important to merit a human fix and to affect the program's test suite.

A candidate subject program is a software project containing at least 50,000 lines of C code, 10 viable test cases, and 300 versions in a revision control system. We acknowledge that these cutoffs are arbitrary rules of thumb; we selected them in the interest of considering systems of non-trivial size and development maturity. We consider all *viable versions* of a program, defined as a version that checks out and builds unmodified on 32-bit Fedora 13 Linux.[4] A program *builds* if it produces its primary executable, regardless of the exit status of **make**.

Testing methodologies vary between projects, to the point that projects may differ on the definition of what constitutes an individual test case. For example, some program test suites are divided by area of functionality tested, with each area breaking down into individualized test suites. Other programs do not delineate their test cases in this way. We define a *test case* to be the smallest atomic testing unit for which individual pass or fail information is available. If a program has 10 "major areas" that each contain 5 "minor tests" and each "minor test" can pass or fail, we say that it has 50 test cases. A *viable test case* is a test that is reproducible, non-interactive, and deterministic in the cloud environment (over at least 100 trials).

We write $\mathit{testsuite}(i)$ to denote the set of viable test cases passed by viable version i of a program. We use all available viable tests, even those added *after* the version under consideration, under the assumption that the most recent set of tests correspond to the "most correct" known specification of the program. We exclude programs with test suites that take longer than one hour to complete in the cloud environment.

We say that a *testable bug* exists between viable versions i and j of a subject program when:

1) $\mathit{testsuite}(i) \subsetneq \mathit{testsuite}(j)$ and
2) there is no $i' > i$ or $j' < j$ with the $\mathit{testsuite}(j) - \mathit{testsuite}(i) = \mathit{testsuite}(j') - \mathit{testsuite}(i')$ and
3) the only source files changed by developers to reach version j were **.c**, **.h**, **.y** or **.l**

The second condition requires a minimal $|i - j|$. The set of specification tests (i.e., that encode required behavior) is defined as $\mathit{testsuite}(i) \cap \mathit{testsuite}(j)$. The specification tests must pass both versions. The witness (i.e., the test cases that demonstrate the bug) is $\mathit{testsuite}(j) - \mathit{testsuite}(i)$. Note that the two sets of tests are disjoint. Note also that the witness can consist of more than one test case, and that we treat all test cases in the witness as corresponding to the same defect (to avoid bias possibly introduced by manually categorizing the failed test cases). Researchers using this dataset could treat scenarios with multiple test cases in the witness either jointly or individually, as appropriate for their application.

Given a viable candidate subject program, its most recent test suite, and a range of viable revisions, we construct a set of testable bugs by considering each viable version i and finding the minimal viable version j, if any, such that there is a testable bug between i and j. We considered all viable revisions appearing before our start date in May, 2011 for all programs besides **php** and **valgrind**; these latter programs include defects appearing before July 2013 as a potential source of testable bugs. We did not cap the number of defects per program to a percentage of the total (as in the initial iteration of this dataset [49]), because we wanted the dataset to support both latitudinal and longitudinal studies (that is, studies of repair success across many programs, and studies of sequential repairs in one program). The latter benefits from a set that contains at least one program with many defects (such as **php** in our dataset). However, we expect that latitudinal studies may benefit from limiting the number of defects from one program (e.g., to under 50% of the total set) to avoid having it dominate or skew the results.

Given these criteria, we canvassed the following sources:

1) the top 20 C programs on popular open-source web sites sourceforge.net, github.com, and Google code,
2) the largest 20 non-kernel Fedora source packages, and
3) programs in other repair papers [32], [52] or known to the authors to have large test suites.

Many otherwise-popular projects failed to meet all of our criteria. Many open-source programs have nonexistent or weak test suites; non-automated testing, such as for GUIs; or are difficult to modularize, build and reproduce on our architecture (e.g., **eclipse**, **openoffice** and **firefox** had test harnesses that we were unable to modularize and script; we were unable to find publicly available test suites for **ghostscript** and **handbrake**). For several programs, we were unable to identify any viable defects according to our definition (e.g., **gnucash**, **openssl**). Some projects (e.g., **bash**, **cvs**, **openssh**) had inaccessible or unusably small version control histories. Other projects were ruled out by our 1-hour test suite time bound (e.g., **gcc**, **glibc**, **subversion**). These projects typically had disk-intensive test suites, and

4. 32-bit Fedora 13 Linux served as a lowest common denominator OS available on the EC2 cloud computing framework as of May, 2011, when we began developing these benchmarks. In the time since, new versions of the operating system have been released, and 64-bit architectures have increased in common usage. Researchers may legitimately prefer to evaluate their techniques in a more modern environment. This is indicative of the tension between keeping a benchmark set current and allowing reproducibility of previous results. We attempt to mitigate this tension with two specially constructed virtual machine images, discussed in Section 4.4.

ManyBugs benchmark							
			test suite				
program	kLOC	defects	count	format	median LOC	stmt coverage	description
`fbc`	97	3	483	BASIC	35	80%	legacy language compiler
`gmp`	145	2	146	C	117	64%	multi-precision math library
`gzip`	491	5	12	Bash	25	34%	data compression utility
`libtiff`	77	24	78	Bash/C	7	25%	image processing library
`lighttpd`	62	9	295	Perl	106	62%	web server
`php`	1,099	104	8,471	phpt	35	80%	web programming language
`python`	407	15	355	Python	203	67%	general-purpose language
`valgrind`	793	15	565	vgtest/C	40	62%	dynamic debugging tool
`wireshark`	2,814	8	63	Bash	—	44%	network packet analyzer
total	**5,985**	**185**	**10,468**				

Fig. 1. The ManyBugs subject C programs, test suites, and historical defects: ManyBugs defects are defined as deviations in program behavior as compared to the next viable version in the history. ManyBugs tests were taken from the most recent version available from the specified start date. This typically corresponds to test case failures fixed by developers in subsequent versions. The test suite columns summarize characteristics of the test suites, further explained in text. Note that `gzip` as configured from its repository differs from that which is released in source packages; the latter includes approximately 50kLOC, depending on the release. The `php` defects are intended to allow studies of iterative repair, or many sequential repairs to one program; an evaluation that investigates generality across many different types of programs may restrict the number of `php` defects considered (e.g., to 45, as we did in previous work).

would arbitrarily slow in the EC2 environment; we anticipate that advancement in virtualization infrastructure will mitigate this limitation, eventually allowing the inclusion of such programs. Earlier versions of certain programs (e.g., `gmp`) require incompatible versions of `automake` and `libtool`. Despite these efforts, we acknowledge that our benchmark set is not exhaustive and will welcome new additions in the future. By formalizing the methodology for generating and reproducing this type of experimental scenario, we hope to encourage community participation and consensus in extending and using the scenarios.

4.2 Constituent programs

Figure 1 summarizes the programs in the ManyBugs benchmark. This section provides details on the programs themselves, to aid in understanding the types of programs in the dataset and to substantiate our claim that the benchmarks support experiments that will generalize to real-world practice. Section 4.3 categorizes and describes the defects.

The ManyBugs benchmark consists of nine well-established open-source programs with mature codebases. These programs are found on many machines running Linux or Mac OSX and are common in the pantheon of open-source software. The benchmark consists of programs developed by both large and small teams. In total, the programs in the benchmark are the result of 285,974 commits made by 1,208 contributors writing 7,745,937 lines of code [12]. The programs in the benchmark are:

1. `fbc` (FreeBASIC Compiler)[5] is a free, open-source multi-platform (though limited to 32-bit architectures as of June 2014) `BASIC` compiler whose codebase covers a large number of common C programming idioms including pointers, unsigned data types, and inline functions. The compiler itself provides a pre-processor and many libraries supporting various compiler extensions. `fbc` has been rated close in speed with other mainstream compilers such as `gcc`.[6]
2. `gmp`[7] is a free open-source library supporting arbitrary precision arithmetic. The library operates on signed integers as well as rational and floating-point numbers. `gmp` emphasizes speed and efficiency, and is intended to target and support cryptography, security applications, and research code.
3. `gzip` (GNU zip)[8] is a data compression utility designed to be a free, superior alternative to `compress`.
4. `libtiff`[9] is a free, open-source library for reading, writing, and performing simple manipulations of Tagged Image File Format (`TIFF`) graphics files. `libtiff` works on 32- and 64-bit architectures on a variety of platforms.
5. `lighttpd`[10] is a lightweight web server optimized for high-performance environments. It is designed to be fast and efficient with a small memory footprint. It is used by `YouTube` and `Wikimedia`, among others. Beyond efficiency concerns, the `lighttpd` project prioritizes compliance to standards, security, and flexibility.
6. `php` (PHP: Hypertext Preprocessor)[11] is a server-side

5. http://www.sourceforge.net/projects/fbc/
6. http://www.freebasic.net/
7. http://www.gmplib.org
8. http://www.gnu.org/software/gzip/
9. http://www.libtiff.org/
10. http://www.lighttpd.net/
11. http://www.php.net/

scripting language designed for web development. The **php** program in this benchmark is the interpreter for the language, which is largely written in C, with a large testing framework and portions of its compilation phase reliant on **php** itself. The defect scenarios in MANYBUGS are restricted to C code, but make use of the bootstrapped testing and compilation framework; the **phpt** language noted in the *test suite* column of Figure 1 refers to a declarative test format used for **php** testing.[12] **php** was not originally intended to be a new programming language, but is now used as such, and has become a critical component of the web (used by Facebook, among others). The **php** language grew organically, leading to some inconsistencies in the language; the development discipline at the project has resulted in a large number of regression tests and a comprehensive bug and feature database.

7. **python**[13] is a widely used general-purpose, high-level programming language. The benchmark program in MANYBUGS is its interpreter, which is written in both C and **python**. The scenarios in the benchmark are restricted to C code.[14] The **python** language emphasizes code readability and extensibility, supporting multiple paradigms such as object-oriented and structured programming among others [77].
8. **valgrind**[15] is a GPL-licensed programming tool for memory debugging, memory leak detection, and profiling. It can be used to build new dynamic analysis tools. Although **valgrind** was originally designed to be a free memory debugging tool for Linux on x86, it has since evolved to become a generic framework for creating dynamic analysis tools such as checkers and profilers.
9. **wireshark**[16] (originally named Ethereal) is a free network packet analyzer. It is used for network troubleshooting, analysis, software and communications protocol development, and education.

The testing methodology varies considerably between projects. The *test suite* columns in Figure 1 provides the number of test cases in each suite, the dominant language used to write those tests, the median non-blank lines of code per test case, and the statement coverage for the modules in which the defects under repair are found. We exclude common configuration or initialization scripts, which each project's testing harness includes, as well as the driver code that execute the tests for a given benchmark, such as the check target in a Makefile. We also exclude median lines of code for **wireshark** because the entire test suite is included in one large, consolidated **bash** script that calls the underlying utilities in turn. The language compilers/interpreters (**fbc**, **php**, and **python**) construct tests out of fragments of the language in question; **php** additionally makes use of a hand-rolled testing framework. **valgrind** also includes its own test framework. **vgtest** files describe the test plans, which typically involve analyzing a particular C file. **lighttpd** uses the **Test::More** Perl test framework. **gmp** and **libtiff** largely construct tests out of C programs that use the library in question.

Test suite size and defective module coverage also vary significantly between the programs. Coverage in particular appears to vary especially by testing approach. Notably, **wireshark**, **libtiff**, and **gzip**'s test suites are probably best characterized as system tests, in that each test calls stand alone programs in various sequences (e.g., the base **libtiff** library consists of many separate utilities). The other programs appear to have a more modular program construction and testing methodology; this is especially true of **php**.

4.3 Categorization

As discussed in Section 3.2, one of our goals in designing these objects of study is to enable qualitative discussion of the success characteristics of various techniques. To this end, beyond characterizing the programs by size and type (described in Section 4.2), we manually categorized each defect in the MANYBUGS set to provide more information about the defects. We expect that results will lend themselves to statements such as "New technique X is well-suited to addressing incorrect output, but does not handle segmentation faults well." We are releasing this categorization with the benchmark scenarios, and anticipate adding new features to the categories as they are identified (either in our experiments or the experiments of others as communicated to us). We provide a high-level description of the categories and categorization process here, but elide features in the interest of brevity; full details are available with the released data and scenarios.

We manually categorized the defects by first searching for developer annotations surrounding the defect. We analyzed bug reports, available in 60 percent of the scenarios, for relevant keywords. We also inspected commit comments in the fix revision, the modification in the human patch, and the witness test case files for insight about the change or comments describing the behavior or feature under test.

The four most common defect categories were:

- 70 instances of incorrect behavior or incorrect output
- 27 segmentation faults
- 20 fatal errors (non-segmentation fault crashes of otherwise unspecified type)
- 12 instances of feature additions

There is some overlap in these categories. For example, crashing errors can be related to segmentation faults, buffer overruns, integer overflows, or a number of other possible causes. Our goal was to label the scenarios as

12. **php**'s use of the **php** language for the testing framework presents a particular challenge in constructing reproducible, idempotent testing scenarios, since test setup, execution, and tear down and cleanup rely on the interpreter working at least in part as expected. The default scenarios as presented here, maintain the default testing framework to remain consistent with the developer intent in its design. We have otherwise worked around the resultant limitations to the best of our ability.

13. http://www.python.org

14. As with **php**, **python**'s build system includes substantial bootstrapping in its compiler, which complicated the construction of reproducible scenarios.

15. http://valgrind.org/

16. http://www.wireshark.org/about.html

informatively as possible for those attempting to either slice the defect set according to their experimental needs, or categorize their results based on defect type. We tried to be as specific as possible with our labeling scheme, and included multiple labels when supported by our investigation. For example, if a particular defect caused a segmentation fault because of a buffer overflow, we label the scenario with both "segmentation fault" and "buffer overflow." We anticipate adding more labels as they are identified by ourselves or other researchers.

We define "incorrect behavior or output" as any instance of the program printing unexpected values or storing unexpected values in memory. For example, PHP bug #61095 is a hexadecimal bug in which $\texttt{0x00} + 2$ incorrectly sums to 4 instead of 2. Some cases of incorrect behavior apply specifically to exceptional situations, such as in **lighttpd** revision #2259, which addressed a defect (bug #1440) in the **secdownload** module that resulted in the error code 410 ("Gone") being returned instead of the appropriate 408 code ("Timeout"). "Feature additions" are bugs that humans addressed by, for example, adding support for a new type definition, adding optional variables to a function, or adding new behaviors to a module. For example, commit # 5252 in the **fbc** compiler added support for octal numbers formatted with &... (where previously just &O was supported). Feature requests are often tracked in the same repository as regular defects, and in several cases we were able to link such scenarios to reports.

We classified defects further along dimensions such as assigned bug priority (when available), whether or not the defect is security related, and wall clock time between bug revision and fix revision. We manually evaluated each developer-provided patch, and note whenever variable types are changed or conditionals are added in the modified code. Finally, we used the **diff** between the bug revision and fix revision to calculate the number of files and the number of lines edited. We elide summary statistics for brevity, but have made this information available as part of the dataset release.

We hope that these categories and annotations will support the qualitative analyses of program repair techniques. For instance, repair techniques that focus on fixing segmentation faults can report results for the subset of defects corresponding to segfaults and possibly other fatal errors. Similarly, repair techniques that limit the scope of their repairs to modifying conditionals can report results for the subset of defects in which the developers modified conditionals in their patch.

4.4 Environment and scenario structure

The ManyBugs scenarios, categorization data, and baseline results and output logs, are available for download: see http://repairbenchmarks.cs.umass.edu/. We give a high-level view of the scenario structure here and provide detailed READMEs, including the categorization details discussed in Section 4.3, with the dataset.

Each ManyBugs scenario is named following a convention indicating the software system name and revision identifiers in question.[17] Each scenario archive includes at least the following components:

- The program source code tree, checked out and configured to build at the defect revision.
- A text file listing the modules or source code implicated in the change (simplifying the slicing of the system's functionality to just the source under repair by the humans, if relevant).
- A test script that takes as arguments a test name and the name of a compiled executable to test. The test names distinguish between initially passing test cases and the witnesses of the bug, and are indexed by number.
- A compile script that takes the name of an executable, corresponding to a folder in which the source code for a program variant is placed.
- A folder containing the version of the implicated files committed at the human-written fix revision.
- A folder containing the patches produced by **diff** corresponding to that human-written change.
- A folder containing the preprocessed C code corresponding to the files implicated in the change (for ease of parsing).
- Sample configuration files for GenProg v2.2, for demonstration and reproduction. These configuration files, with additional arguments, may also be used to initialize AE and replicate the TrpAutoRepair experiments.
- A **bug-info** folder containing data on the defect and tests in the scenario.

Many scenarios include additional support files, primarily intended to support compilation and testing of particular software systems.

We provide two sets of virtual machine images on which we intend the scenarios to be evaluated. While it is possible that the defects at the associated revisions are reproducible in other environments (including in other operating systems), we only make claims and provide explicit reproduction support for the environment we provide. Each set includes an image that can be directly imported into VirtualBox,[18] a free and open-source desktop virtualization software. As of the writing of this article, this format is interoperable with and importable to other popular virtualization options, such as VMWare.[19] The other image is a raw bit-wise dump of the image, which can be imported (with some conversion) into other cloud or virtual environments. We have also created and made public Amazon Machine Images (AMI) that replicate these machines in the EC2 environment. See the documentation associated with the download for details.

The first set of virtual machine images reproduces the 32-bit Fedora 13 environment first used in the evaluation in [49] and to produce the baseline results described below.

17. For software associated with **git** repositories, we include the date of the fix commit in the scenario name to impose an easily-identifiable temporal ordering on the defects in each program set.
18. http://www.virtualbox.org
19. http://www.vmware.com

This represents a reasonable lowest-common denominator for use on Amazon's EC2 cloud computing environment. These images allow direct reproduction of and comparison to previous results (including the baseline results in this article) and trivially enable execution and analysis of the defects that require 32-bit architectures (especially those in **fbc**, which, despite our best efforts and the claims of the associated documentation, we have been consistently unable to compile in a 64-bit environment).

The second set of images reproduces a 64-bit Fedora 20 environment, with a **chroot** jail that replicates the 32-bit Fedora environment. Not all the programs and defects in the MANYBUGS set can be compiled and run directly in a 64-bit environment (though most can). The **chroot** jail allows tools and techniques that require more recent library versions or 64-bit environments to be compiled *outside* the jail and executed *within* it on defects that require the 32-bit environment. Instructions are included with the README associated with the virtual machine images.

INTROCLASS benchmark						
program	LOC	tests		defects		description
		bb	wb	bb	wb	
checksum	13	6	10	29	49	checksum of a string
digits	15	6	10	91	172	digits of a number
grade	19	9	9	226	224	grade from score
median	24	7	6	168	152	median of 3 numbers
smallest	20	8	8	155	118	min of 4 numbers
syllables	23	6	10	109	130	count vowels
total	**114**	**42**	**53**	**778***	**845***	

*The intersection of the 778 black-box and 845 white-box defects is 998 defects.

Fig. 2. The six INTROCLASS benchmark subject programs. The black-box (bb) tests are instructor-written specification-based tests, and the white-box (wb) tests are generated with KLEE to give 100% branch coverage on the instructor-written reference implementation. The 998 unique defects are student-submitted versions that fail at least one, and pass at least one of the tests.

5 THE INTROCLASS BENCHMARK

The INTROCLASS benchmark consists of 998 defect scenarios, and is designed for evaluations that can identify the factors that affect the success of repair techniques. The programs are associated with carefully designed test suites that yield either high specification coverage or 100% branch coverage on a reference implementation. These suites make this benchmark particularly suitable for evaluating the effects of test suite quality, coverage, and provenance on the repair. The varied severity of the defects, in terms of the number of test cases they cause the program to fail, makes these scenarios particularly suitable for evaluating the effect of initial program quality on repair. As this benchmark is composed of small programs, it can also be used to evaluate early-stage techniques that do not yet scale to programs of larger size or complexity.

This section discusses the methodology we used to construct the INTROCLASS benchmark (Section 5.1), the process for creating the two test suites for each benchmark program (Section 5.2), the methodology for using the test suites to identify the defects (Section 5.3), the details of each of the benchmark programs (Section 5.4), and the instructions on how to download, run, and otherwise use the programs (Section 5.6).

The INTROCLASS benchmark may evolve. This section and the results in Section 6 describe MANYBUGS v1.0.

5.1 Methodology

The INTROCLASS dataset is drawn from an introductory C programming class (ECS 30, at UC Davis) with an enrollment of about 200 students. The use of this dataset for research was approved by the UC Davis IRB (intramural human studies review committee), given that student identities are kept confidential. To prevent identity recovery, students' names in the dataset were salted with a random number and securely hashed, and all code comments were removed.

The programming assignments in ECS 30 require students to write C programs that satisfy detailed specifications provided by the instructor. For each assignment, up to the deadline, the students may submit their code at any time, and as many times as they want; there is no penalty for multiple submissions. Each time they submit, they receive a notification about how many tests their program passed, but they see neither the test inputs nor the test outputs. The students then may rethink the specification, their implementations, and resubmit. The students also have access to an oracle, which they may query with inputs and receive the expected output. A portion of the grade is proportional to the number of tests the program passes.

For each of the six assignments (see Figure 2), each time the students submit a potential solution, this solution is recorded in a **git** repository unique to that student and that assignment. The INTROCLASS benchmark consists of 998 submitted solutions, each of which fails at least one and passes at least one black-box test or fails at least one and passes at least one white-box test (Section 5.2 describes the test suites, and Section 5.3 describes the defects.) Further, the benchmark includes the final student solutions, many (but not all) of which pass all the tests.

5.2 Test suites

For each benchmark program, we developed two test suites: a black-box test suite and a white-box test suite. These test suites are intended to be different while each covering the program's behavior.

The *black-box test suite* is based solely on the program specification and problem description. The course instructor constructed this test suite manually, using equivalence partitioning: separating the input space into several equivalent partitions, based on the specification, and selecting one input from each category. For example, given a program that computes the median of three numbers, the black-box test suite contains tests with the median as the first, second, and third input, and also tests where two and all three inputs are equal.

The *white-box test suite* is based on the oracle program for each assignment, whose output is by definition correct for the assignment in question. The white-box test suite achieves branch coverage on the oracle. Whenever possible, we create the white-box test suite using KLEE, a symbolic execution tool that automatically generates tests that achieve high coverage [18]. When KLEE fails to find a covering test suite (typically because of the lack of a suitable constraint solver), we construct a test suite manually to achieve branch coverage on the oracle.

The black-box and white-box test suites are developed independently and provide two separate descriptions of the desired program behavior. Because students can query how well their submissions do on the black-box tests (without learning the tests themselves), they can use the results of these tests to guide their development.

5.3 Defects

We evaluated the student-submitted program versions (corresponding to potential assignment solutions) against each of the two test suites. We identify *subject defects* by considering every version of every program that passes at least one test and fails at least one test, for each test suite. We exclude versions that fail all tests because of our requirement that benchmark scenarios conform at least in part to a (partial) correctness specification. Program versions that fall all tests in a test suite are typically so malformed that they are too different from a correct solution to be considered a defect.

As Figure 2 summarizes, we identified a total of 778 defects using the black-box test suite, and 845 defects using the white-box test suite. The intersection of these two sets is 998 defects. Of course, some students may have made similar mistakes and introduced similar, or even identical bugs in their code. Because the INTROCLASS benchmark is representative of both the type and the frequency of bugs made by novice developers, we did not remove duplicates from our dataset; however, some uses of the dataset may require first identifying and then removing these duplicates.

5.4 Constituent programs

Figure 2 summarizes the programs in the INTROCLASS benchmark. This section provides details on the programs themselves, to help users of the benchmark understand the types of programs available for their evaluation.

The INTROCLASS benchmark consists of six small C programs. These programs can be compiled by `gcc` with default settings. The programs are:

1. **`checksum`** takes as input a `char*` single-line string, computes the sum of the integer codes of the characters in the string, and outputs the `char` that corresponds to that sum modulo 64 plus the integer code for the space character. The black-box test suite includes inputs strings of all lower-case letters, upper-case letters, numbers, special characters, and combinations of those types of characters.
2. **`digits`** takes as input an `int` and prints to the screen each base-10 digit appearing in that input, from the least significant to the most significant, in the order in which they appear. Each digit is to be printed on a separate line. The black-box tests include positive and negative inputs, inputs with single and with multiple digits, and inputs consisting of a repeated digit.
3. **`grade`** takes as input five `double` scores. The first four represent the thresholds for A, B, C, and D grades, respectively. The last represent a student grade. The output is the string "`Student has a X grade\n`", with "X" replaced by the grade based on the thresholds. The black-box test suite includes a student grade that falls in each of the ranges defined by the thresholds, outside of the ranges, and on each boundary.
4. **`median`** takes as input three `int`s and computes their median. The students are asked to use as few comparisons as possible, and are told that it is possible to produce a correct program that performs only three comparisons. The black-box test suite includes sets of numbers such that each of the positions is the median, and sets of numbers with two and with three identical numbers.
5. **`smallest`** takes as input four `int`s, computes the smallest, and prints to the screen "`X is the smallest`" where "X" is replaced by the smallest `int`. The students are asked to use as few comparisons as possible. The black-box test suite includes orderings of the four inputs such that the minimum falls in each of the positions, positive and negative numbers, and sets that include three or four identical integers.
6. **`syllables`** takes as input a `char*` string of 20 or fewer characters and counts the number of vowels (a, e, i, o, u, y) in the string. The program should print to the screen "`The number of syllables is X.`", where "X" is replaced by the number of vowels. The black-box test suite includes strings with spaces, special characters, as well as zero and one vowel.

5.5 Categorization

As with the MANYBUGS suite, we categorized the defects in the INTROCLASS suite. For each of the 998 code versions from Figure 2, we ran each test and observed the code's behavior on that test. There were a total of 8,884 test failures. The overwhelming majority of the failures were caused by incorrect output. The causes of the failures were:

- 8,469 instances of incorrect output
- 85 timeouts (likely infinite loops)
- 76 segmentation faults
- 254 other (non-segmentation fault) exit status errors

The metadata file for each defect (see Section 5.6 and Figure 3) includes the failure cause for each failing test.

```
checksum
  tests
    blackbox
      1.in
      1.out
      2.in
      2.out
      ...
    whitebox
      1.in
      1.out
      2.in
      2.out
      ...
    checksum.c   (instructor-written solution)
    checksum     (compiled instructor-written solution)
  studentIDhex_1
    commitID_1
      blackbox_test.sh
      whitebox_test.sh
      checksum.c
      ae-bb-01.log
      ae-wb-01.log
      gp-bb-01.log
      ...
      gp-bb-20.log
      gp-wb-01.log
      ...
      gp-wb-20.log
      trp-bb-01.log
      ...
      trp-bb-20.log
      trp-wb-01.log
      ...
      trp-wb-20.log
      metadata.json
      Makefile
    commitID_2
      ...
    commitID...
    Makefile
  studentIDhex_2
    ...
  studentIDhex...
  Makefile
digits
  ...
grade
  ...
median
  ...
smallest
  ...
syllables
  ...
Makefile
README
defect-classification.json
```

Fig. 3. File structure of the IntroClass benchmark. The hex labels are anonymized student id hashes and the `metadata.json` file in each program version directory contains metadata on the tests that version passes and fails, the test execution outputs, and if the version is nondeterministic. Each commitID directory contains one defect; the largest commitID is the final student-submitted version, which usually passes all tests, but sometimes is a defect. The `.log` files are the execution logs of the repair tools. Due to nondeterminism, the repair tools failed to run on some defects (e.g., because a test expected to fail sometimes passed), so no `.log` are reported for tool executions on those defects. The final student-submitted versions that pass all tests do not have `.log` files they are not defects.

5.6 Environment and scenario structure

The IntroClass download package contains the instructor-written (correct) programs, the defects, the white- and black-box test suites, execution infrastructure test scripts, makefiles for compiling the programs, and program metadata.

Figure 3 describes the structure of the downloadable IntroClass package. Each defect's metadata file lists which tests that program version passes and fails, and contains the version's outputs on each of the test cases. Finally, each defect contains two shell test scripts (one for white-box testing and one for black-box testing), with an interface analogous to the one described in Section 4.4. To build the IntroClass dataset, run GNU `make` in the top-level directory, or in any subdirectory (to build a subset of the dataset).

6 Empirical results

In this section, we report the results of running GenProg v2.2, TrpAutoRepair, and AE v3.0 on all of our defect scenarios (as discussed in Section 3.3). We remind the reader that the purpose of this article is not to evaluate GenProg, TrpAutoRepair, and AE or any other particular repair technique. Other work, e.g., [17], [49], [71] has used these datasets or parts of these datasets to evaluate techniques. Instead, our goal is to provide a useful benchmark suite for the subfield of automated program repair. Thus we neither examine nor discuss the results in detail, but instead provide them here to save researchers time, money, and compute resources; to admit rapid comparisons against previous techniques when new tools are developed; and to demonstrate by example the use of these benchmark suites for the purposes of evaluating state-of-the-art techniques in automatic repair. Note that these baseline results are suitable for comparison only when the programs, test suites, and environments are kept the same; reruns will still be necessary otherwise. We have released these results with the dataset. We anticipate that they could serve to highlight categories of programs and defects that are not well-handled by the current state of the art, focusing future research efforts.

	GenProg			TrpAutoRepair			AE		
Program	Defects repaired	Time (min)	Fitness evals	Defects repaired	Time (min)	Fitness evals	Defects repaired	Time (min)	Fitness evals
`fbc`	1/3	133	79.0	0/3	—	—	1/3	7	1.7
`gmp`	1/2	13	7.2	1/2	18	2.4	1/2	739	63.3
`gzip`	1/5	240	130.7	1/5	107	56.7	2/5	84	1432
`libtiff`	17/24	27	20.8	17/24	16	2.9	17/24	24	3.0
`lighttpd`	5/9	79	44.1	4/9	33	14.9	4/9	22	11.2
`php`	54/104	181	5.2	56/104	180	1.1	53/104	441	1.1
`python`	2/15	110	12.9	2/15	144	1.4	3/15	529	7.6
`valgrind`	4/15	193	24.0	4/15	133	1.5	0/15	—	—
`wireshark`	5/8	140	14.3	5/8	44	2.6	5/8	574	66.5

Fig. 4. MANYBUGS: Baseline results of running GenProg v2.2, TrpAutoRepair, and AE v3.0 on the 185 defects of the MANYBUGS benchmark. For each of the repair techniques, we report the number of defects repaired per program; the average time to repair in minutes (GenProg and TrpAutoRepair were run on 10 seeds per scenario, with each run provided a 12-hour timeout; AE is run once per scenario, with a 60-hour timeout); and the number of fitness evaluations to a repair, which serves as a compute- and scenario-independent measure of repair time (typically dominated by test suite execution time and thus varies by test suite size). Complete results, including individual log files for each defect, are available for download with the dataset.

6.1 Experimental setup

These baseline results are based on experimental parameters using the latest published methodology for each algorithm. We used the off-the-shelf GenProg v2.2 and AE v3.0, from the GenProg website (http://genprog.cs.virginia.edu). TrpAutoRepair [66] is described by its authors as extending the publicly available GenProg codebase, although the novel features described for TrpAutoRepair (notably the test suite prioritization technique) were independently developed in AE [75]. Since the TrpAutoRepair prototype is not publicly available, we reimplemented the TrpAutoRepair algorithm based on its publication [66].

Because both GenProg and TrpAutoRepair are randomized algorithms, each effort to repair a given scenario consists of a number of random trials run in parallel. For MANYBUGS, we performed 10 such trials (following their published descriptions); for INTROCLASS, which is much less compute-intensive to evaluate, we performed 20 to admit additional statistical confidence. For GenProg and TrpAutoRepair, each MANYBUGS trial was given up to 12 hours or 10 generations, whichever came first, again following published experimental methodologies; each INTROCLASS trial was given 100 generations. Timing results are reported for the first of the trials to succeed for each scenario. Full GenProg parameters are provided with the debug logs of these results. As high-level guidance and following reported values for these algorithms, we set the population size to be 40, mutated each individual exactly once each generation, performed tournament selection, and applied crossover once to each set of parents. Because of the compute costs involved in fitness evaluation on large test suites, for MANYBUGS, the fitness function samples 10% of the passing test suite for all benchmarks for the evaluation of intermediate candidate solutions; if a solution passes the full 10% of the test suite as well as the initially failing test cases (encoding the defect under repair), the variant is tested on the remaining 90% to determine if it represents a candidate solution. We did not sample for INTROCLASS. For both datasets, we use the fault space, mutation operation weighting, and other choices described in the most recent large study of GenProg effectiveness [50].

TrpAutoRepair uses random search and a test suite selection strategy and is thus not generational. In keeping with its published parameters, each trial consisted of 400 random individuals, and we otherwise matched the random weighting parameters provided to GenProg. Test suite sampling does not apply to TrpAutoRepair.

Because AE is not randomized, each effort to repair a given defect scenario consists of a single trial with the edit distance k set to 1; AE has no notion of population or fitness. Each AE trial was given up to 60 hours to complete for MANYBUGS, and 25 minutes for INTROCLASS.

For the MANYBUGS experiments, we used Amazon's EC2 cloud computing infrastructure with our Fedora 13 virtual machine image. Each trial was given a high-cpu medium (c1.medium) instance with two cores and 1.7 GB of memory. [20] For the INTROCLASS experiments, we used Ubuntu 13.10 double-extra large high CPU instances (c3.2xlarge) with eight cores and 15 GB of memory. [21]

6.2 Baseline results on MANYBUGS and INTROCLASS

Figure 4 shows the results of executing GenProg v2.2, TrpAutoRepair, and AE v3.0 on the 185 defects in the MANYBUGS dataset. (The unified results reported here differ slightly from previously published results for **`php`** and **`wireshark`** [49] due to the higher standards in this work for reproducible, deterministic test cases.) Figure 5 shows the

20. https://aws.amazon.com/ec2/previous-generation/
21. http://aws.amazon.com/ec2/instance-types/

Program	GenProg Defects repaired	GenProg Time (sec)	GenProg Fitness evals	TrpAutoRepair Defects repaired	TrpAutoRepair Time (sec)	TrpAutoRepair Fitness evals	AE Defects repaired	AE Time (sec)	AE Fitness evals
white-box-based defects									
`checksum`	3/49	343	132	1/49	10	5	1/49	4	1
`digits`	99/172	191	102	46/172	32	13	50/172	11	3
`grade`	3/224	152	160	2/224	26	23	2/224	25	25
`median`	63/152	107	114	36/152	19	25	16/152	4	2
`smallest`	118/118	23	23	118/118	15	11	92/118	4	2
`syllables`	6/130	284	157	9/130	36	56	5/130	9	6
black-box-based defects									
`checksum`	8/29	517	307	0/29	—	—	0/29	—	—
`digits`	30/91	162	77	19/91	24	15	17/91	6	6
`grade`	2/226	141	156	2/226	30	27	2/226	24	25
`median`	108/168	44	59	93/168	20	20	58/168	4	1
`smallest`	120/155	102	86	119/155	24	21	71/155	5	4
`syllables`	19/109	96	117	14/109	39	54	11/109	3	2

Fig. 5. INTROCLASS: Baseline results of running GenProg v2.2, TrpAutoRepair, and AE v3.0 on the 845 white-box-based defects, and 778 white-box-based defects of the INTROCLASS benchmark. For each of the repair techniques, we report the number of defects repaired per program; the average time to repair in second (all three techniques were given timeouts); and the number of fitness evaluations needed to produce a repair. Complete results, including individual log files for each defect, are available for download with the dataset.

same baseline results on the INTROCLASS dataset. For each of the techniques, we report the number of defects repaired per program out of the total scenarios per program, with the parameters described in Section 6.1. Following the importance of efficiency in real-world program repair, we report wall-clock time to repair on average across all scenarios repaired for a program. We also present the average number of test suite executions ("fitness evaluations" in the figures) in runs leading to a repair. This measurement serves as a compute- and scenario-independent measure of efficiency, which is typically dominated by test suite execution time.

There are many other interesting measurements that we could report, such as monetary cost on public cloud compute resources (which we omit because it serves as a proxy for compute time), patch size, patch complexity or readability, etc. We choose these baselines based on their ubiquity in previous studies of repair success and their relationship to the real-world use case of automatic repair. However, we believe there is much work to be done in defining new and more comprehensive measurements of repair efficiency and quality, in particular. Because such patches will likely be maintained and thus must be understood by human developers, good quality measurements are an important area of current and future research.

An in-depth discussion of these results is outside the scope of this article; our goal is to provide baselines and illustrate the potential use of the benchmarks. However, a few observations are worth discussing as part of that illustration. As expected, TrpAutoRepair and AE are generally faster than GenProg. On MANYBUGS, the three techniques repair roughly the same defects, with only minor variation (for example, with respect to which `php` defects are repaired). Even though the techniques vary in several crucial details — determinism, how test cases are used, whether edits can be combined — their common operators may control which defects they are broadly able to tackle. GenProg succeeds more often than TrpAutoRepair and AE on INTROCLASS, suggesting that multi-edit repairs may be particularly important on these smaller programs. On the other hand, AE (and, to a much lesser extent, TrpAutoRepair) is much faster: it repairs only half as many defects, but does so in roughly one-tenth the time. The two search approaches — deterministic vs. stochastic — may thus represent different tradeoffs in the design space.

We provide with the downloadable dataset complete results, including individual log files for each defect, which can be used to reconstruct the produced patches. We encourage other researchers to use these results in support of the generation of new metrics.

7 RELATED WORK

Automated Program Repair. The subfield of automated program repair is concerned with automatically bringing an implementation more in line with its specification, typically by producing a patch that addresses a defect. Since 2009, interest in this subfield has grown substantially, and currently there are at least twenty projects involving some form of program repair (e.g., AE [75], AFix [44], ARC [14], Arcuri and Yao [9], ARMOR [20], and AutoFix-E [73], [61], Axis [54], BugFix [41], CASC [78], ClearView [63], Coker and Hafiz [22], Debroy and Wong [25], Demsky and Rinard [26], FINCH [59], GenProg [49], Gopinath *et al.* [36], Jolt [19], Juzi [31], MintHint [46] PACHIKA [23], PAR [47], SemFix [58], Sidiroglou and Keromytis [69], TrpAutoRepair [66], etc.). While there are multiple ways to categorize program repair approaches, one of the most salient distinctions is between what we call *generate-and-validate* techniques (which heuristically produce multiple candidate repairs for a given bug and then validate each candidate for acceptability) and *correct-by-construction* techniques (in which the particular problem formulation or constraints employed lead to repairs that are provably correct in a mathematical sense). We also distinguish between general or generic repair approaches (which target arbitrary software engineering defects) and defect-specific or targeted repair approaches (which focus on one particular class of bugs). While many correct-by-construction or defect-specific techniques employ implicit specifications (e.g., AFix addresses single-variable atomicity violations, which are assumed to be undesired; Coker and Hafiz fix integer bugs, such as overflow, which are also assumed to be undesired) the majority of current repair approaches target general defects and thus require a specification of desired behavior as well as evidence of the bug. This specification typically takes the form of formal specifications or annotations (as in AutoFix-E), test cases (as in GenProg) or various hybrids (as in SemFix, where annotations are derived from test cases). Several of the more formal repair approaches, particularly those that are correct-by-construction, share commonalities with and are informed by advances in program synthesis, such as component-based [42] and test-driven program synthesis [62]. Such approaches, particularly those that rely on input-output examples or test traces, may also benefit from a standard set of programs with test cases (particularly those that include feature additions). Despite the common bond of operating on off-the-shelf programs with test cases, very few papers in this area compare directly against other techniques, with a few notable exceptions [47], [58], [66]. A few techniques are designed as interactive or developer assistance tools (e.g., Jolt, MintHint, and BugFix), though the underlying principles are consistent with the others.

Novel automated program repair approaches serve as the key client or target application for this article and its benchmark suites. The explosion of program repair approaches, most of which address general defects and can make use of test cases, suggests that this effort is well-timed. This suggests at a high level that our proposed benchmarks may be sufficiently broadly applicable to unify some subset of evaluation of new techniques moving forward. We discuss potential benchmark applicability to the evaluations of these previous techniques in Section 8.

We believe the subfield is mature enough now that new approaches can profitably compare themselves against others in the field, rather than inventing entirely new (and thus incomparable) specialized defect scenarios. While automated program repair shows great promise, it is far from being accepted commercially: many potential improvements remain to be made. In that regard, this article and benchmark set are a response to the adage, "you cannot improve what you cannot measure," in the research domain of program repair.

Software engineering benchmarks. Sim *et al.* issued a call to arms to the software engineering community by presenting case studies of research communities for which a common benchmarking methodology served to build consensus and drive research forward. They argued that "The technical progress and increased cohesiveness in the community have been viewed as positive side-effects of benchmarking. Instead, [they] argue that benchmarking should be used to achieve these positive effects" [70]. We follow this reasoning in presenting the ManyBugs and IntroClass datasets, and hope that doing so will have positive effects on research in automatic bug repair.

Computer science research in general and software engineering in particular have produced several datasets that are well-suited to the study and comparison of particular problem areas. The first set, including SPEC, ParSec, and DaCapo, were designed for performance benchmarking and do not contain the intentional semantic defects required for the automated repair problem. The SPEC benchmarks aim to "produce, establish, maintain and endorse a standardized set" of performance benchmarks for computing [72]. The ParSec suites serve similar purposes, focusing specifically on multi-threaded programs and chip multiprocessors [10]. The DaCapo suite supports performance evaluation of compilation and dynamic analyses like garbage collection for Java programs, designed specifically to address failings in the SPEC suite with respect to Java [13].

Other researchers have developed benchmark methodologies and suites to evaluate bug finding techniques; these are closer in spirit to the ManyBugs and IntroClass suites because they by definition consist of programs with defects. These datasets have each proved useful, but none of them addresses all of the concerns that motivated our benchmarks.

Bradbury *et al.* constructed a suite to evaluate techniques that test or analyze concurrency programs; their argument is similar to the one we present here, namely, that comparison between techniques requires consensus on suitable objects of study. While our datasets focus on deterministic bugs, Bradbury *et al.*'s methodology serves as a useful starting point for corresponding evaluations of nondeterministic defects.

BugBench proposed a suite of programs and defects for use in evaluating bug finding techniques [55]. As we do, they characterize the space of research in automated bug finding and use their conclusions to guide the selection of bugs to include in the suite. Their use case (bug detection) is close to but not identical to our own (bug repair). We focus on the origin of the specifications of both correct and incorrect behavior to ensure that the ManyBugs benchmark is indicative of real-world defects; the test suites in BugBench are constructed largely by hand by the authors, which is compatible with their evaluation goals, but not with all of ours. The iBugs suite and methodology [24] is close in spirit to ManyBugs. The authors propose to construct datasets to evaluate bug finding and localization techniques by mining repositories for key words that indicate revisions at which a bug is repaired. By contrast, we look specifically for revisions at which observed dynamic behavior on the test suite (serving as the specification of desired behavior) changes. In their case, this means that the iBugs scenarios do not always contain test cases for a defect. By construction, our scenarios always do. We argue that a witness test case and a partial specification (e.g., positive test cases) are required for bug scenarios that are suitable for evaluating most existing automatic repair techniques. Defects4J [45] follows a similar methodology, again on Java programs, intended for evaluation of testing techniques.

Perhaps the most common set of programs with defects is from the SIR or Siemens suite [39], [27]. The Siemens suite does provide programs with test suites and faults. However, as with the other existing suites, it was designed with a distinct purpose: to support controlled testing and evaluation of software *testing* techniques. Thus, the test suites are all constructed, and most of the faults are seeded.

It is possible to evaluate repair methods using many of these suites, and likely on subsets of all of them. However, they each lack support for one or more key aspects of common comparative evaluations. Although ManyBugs and IntroClass are certainly not complete nor perfect for any possible evaluation, we constructed them specifically with the needs of the community in mind. Reproducibility is a core concern, and we present a methodology for developing benchmarks, which can support extensions as new programs and techniques arise with new complications.

An initial version of the ManyBugs benchmark suite was described in previous work [49]. That publication reported on a large-scale systematic evaluation of one specific repair method, and it introduced a set of 105 bugs to facilitate that evaluation. This article focuses on the benchmarks and associated methodology specifically; it extends the original ManyBugs dataset significantly, adds the IntroClass dataset, improves several scenarios to catch degenerate behavior, categorizes all of the bugs, and formalizes the methodology.

Many software engineering benchmarks involve the use of seeded defects. We have previous experience applying GenProg to a subset of the Siemens benchmark suite [68, Tab. 1] as well as to seeded defects [68, Sec. 5] following an established defect distribution [34, p. 5] and fault taxonomy (e.g., "missing conditional clause" or "extra statement"). In both cases, GenProg behaved similarly on seeded defects and on real-world defects. There exists both direct and indirect evidence that seeded defects can be as useful as real defects in some settings. For example, some seeded defects are as difficult to locate as real defects [48]. Just *et al.* found both that there exists a correlation between mutant detection and real fault detection by test suites, and that mutation strategies and operators could be improved [45]. Seeded defects and mutations are also studied in the mutation testing community [43]; see Weimer *et al.* [75, Sec. VI] for a discussion of the duality between mutation testing and program repair. Fault seeding might therefore serve as a suitable mechanism for generating defect scenarios of intermediate complexity, between the small, student-written programs of IntroClass and the larger legacy C programs of ManyBugs. However, there is currently no direct evidence that seeded defects are suitable for evaluating automated program repair.

8 Benchmark Generality

No benchmark set is perfect. Consensus benchmarks enable large-scale experiments, replication of results, and direct comparison between techniques. However, as case studies, they enable limited control, which reduces their generalizability. We mitigate this threat by providing the IntroClass dataset, which is designed to allow controlled experimentation. Moreover, performance on a benchmark set typically does not produce results that are amenable to the construction of explanatory theories [70]. In general, having a benchmark may cause researchers to over-optimize their techniques for the benchmark, thus reducing generalizability of the techniques. Further, the relative ease of using a benchmark may unintentionally reduce evaluations that focus on important but harder-to-evaluate measures. For example, in the field of fault localization, researchers optimized for finding localizing faults without, for a long time, considering if perfect fault location information would be of use to developers [60]. This concern is particularly important for automated program repair, because, as discussed in Section 3.1, test cases are flawed as partial correctness specifications. We hope researchers who use these benchmarks will perform multiple types of qualitative analysis and present multiple sources of evidence to support their claims, and that we will continue as a field to develop better ways to measure repair quality overall. Comparative empirical evaluations are important to high-quality empirical science, but we do not believe they are the only source of important evidence to support claims about automatic program repair.

There exist other open-source C programs that could be included in the ManyBugs set, as well as additional defects farther back in the history of some of the programs that, given additional time and resources, could be identified. Our benchmarks may not be as general, nor as indicative of real-world defects, as we claim. Because we do not know the true distribution of bugs in legacy programs, it is difficult to assess the degree to which results on these defects, however systematically identified, can be generalized to the true population of defects in open-source C programs. Our requirement that test suites be modularizable, deterministic, and scriptable likely limits the types of programs that can be included. We mitigate the threat to generalizability and scientific utility by including a broad set of program types and sizes, by categorizing the defects, and by formalizing the methodology by which we constructed both INTROCLASS and MANYBUGS. We hope this will allow principled development of the datasets to follow advances in compute resources and software. The categorization of defects will allow researchers to identify the subsets of defects on which they evaluate and on which their techniques succeed. We plan to incorporate additional labeling schemes when they become available and to either repair or flag problematic scenarios when reported by users.

As software systems continue to grow and as programming and educational paradigms shift, the defects contained in the datasets may become less indicative of those that appear in the wild (as with MANYBUGS), or the types of mistakes that novice programmers make (as with INTROCLASS). By following the examples set by SPEC [72], SIR [27], and others, these benchmarks can be maintained and continually improved, both by the authors and the community of researchers in automated repair. We also constructed a virtual machine image that emulates a current operating system, with support for emulating the original virtual machine. This shows how empirical experiments and testbeds can evolve with technology while still providing backwards compatibility.

It is possible that the scenarios were tailored too narrowly to our own use cases as researchers who study automatic repair. We mitigated this threat by using them to evaluate three different techniques using several success metrics, and by providing two quite different datasets. Our collaboration includes both authors of the tool that initiated our benchmarking effort (GenProg) and authors who have used and evaluated that tool independently with very different experimental purposes. Further, TrpAutoRepair [66] was developed by authors not involved in the benchmarking project. They evaluated on a subset of an earlier version of the defects and used another subset in a focused study of fault localization for program repair [67]. Our combined and diverse experience with the research problems presented by automatic program repair research helps to mitigate the risks of single-project myopia in experimental design. Similarly, the fact that other researchers have already adapted previous and impoverished versions of the scenarios provides evidence of general utility.

Despite our best efforts, it is possible that the scenarios will not prove suitable for evaluating a completely novel program repair techniques in the future, limiting their utility. To help assess this risk, we examined the program repair papers cited in this article (attempting to be exhaustive, although the field is moving quickly and this list is simply our best effort). More than half of the papers are specialized to C programs or are language independent; and all required test cases or workloads at some stage. Based on these criteria, MANYBUGS or INTROCLASS or both (depending largely on scalability, which we could not always judge) could have been used to evaluate TrpAutoRepair [66]; SemFix [58]; Debroy and Wong [25]; He and Gupta [38], whose technique requires test suites with high structural coverage and targets a subset of C, suggesting INTROCLASS could be suitable; Coker and Hafiz [22], who used historical human repairs (which we provide) in their evaluation; BugFix [41]; MintHint [46]; and CASC [78]. Similarly Jolt [19] and ClearView [63] are language independent, address defect types that are included in either or both of the defect sets, and require indicative or runtime workloads.

The remaining papers focus on specific defect classes, or use cases that are not well represented in our datasets [26], [69], or involve non-C languages. The most common such language was Java (PAR [47], Juzi [31], Gopinath *et al.* [36], Axis [54], AFix [44], and FINCH [59]). A diverse set of other languages round out the remainder of the work we surveyed: database selection statements [35], Eiffel [73], [61], Javascript [20], and a demonstration language [7].

The vast majority of the reported techniques, regardless of language, require test cases, and well over half target C. Note that our benchmark construction methodologies are not language-specific. This may allow cross-language technique comparisons for certain types of empirical claims, such as the proportion of a historic set of open-source defects a technique can address [49]. Defects4J [45], for example, is constructed via a process similar to the one we use to produce MANYBUGS. Defects might be sliced by the amount or type of code that must be changed to address them, enabling the identification of a "core" set of defect types common to, e.g., imperative languages, further enabling cross-language defect and repair technique comparisons. However, cross-language comparisons are likely to remain limited: Is it meaningful to compare a Java-specific technique to one that targets C pointer errors? Ultimately, we expect that different languages will benefit from language-specific benchmark construction methodologies. See, for example, the DaCapo suite, which addressed methodological problems with SPEC's direct translation of benchmark construction methodologies for C or Fortran to Java [13]. Although our defect sets certainly cannot apply to all program repair research, the trends we observe in the current state of the field suggests that the general use case we adopted for MANYBUGS and INTROCLASS covers a broad swath of the current research activity in the space of program repair.

We encountered many unanticipated challenges to constructing robust defect scenarios while preparing MANYBUGS and INTROCLASS for public release, and it is unlikely that we found every inconsistency. In particular, MANYBUGS test suites were adapted from those provided by the developers of the subject programs, which may mask degenerate behavior in ways that we did not detect. We mitigated this threat by conducting numerous external checks for possibly degenerate behavior, through multiple sanity checks on each scenario and each potential source of nondeterministic behavior, and we noted other sources of known nondeterminism.

We address other unforeseen sources of nondeterminism, unexpected behavior, or otherwise incorrect analysis through a public release of our datasets and log files for the experiments described in this article. We encourage other researchers to contact the authors with discrepancies they discover or suggestions for improvement, and we will endeavor to keep them and the results as up to date as possible in the face of new information. Finally, we assigned version numbers to each benchmark and associated virtual machines, which we will update as we make changes to the underlying artifacts.

9 Contributions

Research fields rarely begin with standardized methods or evaluation benchmarks. This absence can, at times, empower both creativity and productivity of research. However, as we argued earlier, common benchmarks become important as a field matures to enable researchers to properly reproduce earlier results, generalize, and to compare, contrast, and otherwise evaluate new methods as they come along. Historically, in computer science, the creation of standard benchmarks has led to principled evaluations and significant progress in associated fields, e.g., [72], [10], [13].

To that end, we developed two collections of defect scenarios — ManyBugs and IntroClass — consisting of 1,183 defects in 15 C programs. These benchmarks are diverse and allow for comparative evaluation of different types of automatic repair algorithms and experimental questions. Each program includes multiple defects, test suites, and human-written fixes. We illustrated the use of the benchmarks with three repair techniques, GenProg, TrpAutoRepair, and AE to provide baseline data that can be used in future studies. One of our goals in creating these benchmarks was to enable a broad set of use cases for automatic program repair researchers. We hope that the benchmarks will be useful, even for methods that don't require all of the included components. For example, a technique might not explicitly require a set of initially passing test cases, or an evaluation might not compare to the human-provided patches. We devoted a significant amount of effort to enhance usability and experimental reproducibility, with the goal of increasing the benchmarks' longterm contribution to the research community.

The scenarios, data (including baseline output and categorization information) and virtual machine images are available at http://repairbenchmarks.cs.umass.edu/. The web site provides detailed README files explaining the structure of the package, scenarios, results, and virtual machines information (including AMI numbers for the EC2 images). Errata and feedback on these resources should be sent to the authors.

Acknowledgments

The authors gratefully acknowledge the partial support of AFOSR (FA9550-07-1-0532, FA9550-10-1-0277), DARPA (P-1070-113237), DOE (DE-AC02-05CH11231), NSF (CCF-0729097, CCF-0905236, CCF-1446683, CNS-0905222), and the Santa Fe Institute. In addition, Martin Rinard provided insightful discussions regarding repair quality and identified and corrected several concerns in the defect scenarios.

References

[1] http://www.mozilla.org/security/bug-bounty.html, Accessed Feb, 2014.

[2] http://blog.chromium.org/2010/01/encouraging-more-chromium-security.html, Accessed Feb, 2014.

[3] http://msdn.microsoft.com/en-us/library/dn425036.aspx, Accessed Feb, 2014.

[4] http://msdn.microsoft.com/en-us/library/dn425036.aspx, Accessed Feb, 2014.

[5] http://www.daemonology.net/blog/2011-08-26-1265-dollars-of-tarsnap-bugs.html, Accessed Feb, 2014.

[6] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[7] Andrea Arcuri. On the automation of fixing software bugs. In *Doctoral Symposium — International Conference on Software Engineering*, 2008.

[8] Andrea Arcuri and Gordon Fraser. On parameter tuning in search based software engineering. In *International Symposium on Search Based Software Engineering*, pages 33–47, 2011.

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

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[11] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar T. Devanbu. Fair and balanced? Bias in bug-fix datasets. In *Foundations of Software Engineering*, pages 121–130, 2009.

[12] Black Duck Software, Inc. http://www.ohloh.net/, February 2014.

[13] Stephen M. Blackburn, Robin Garner, Chris Hoffman, Asad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[14] Jeremy S. Bradbury and Kevin Jalbert. Automatic repair of concurrency bugs. In *International Symposium on Search Based Software Engineering — Fast Abstracts*, pages 1–2, Sept. 2010.

[15] Jeremy S. Bradbury, Itai Segall, Eitan Farchi, Kevin Jalbert, and David Kelk. Using combinatorial benchmark construction to improve the assessment of concurrency bug detection tools. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD 2012, pages 25–35, 2012.

[16] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[17] Yuriy Brun, Earl Barr, Ming Xiao, Claire Le Goues, and Prem Devanbu. Evolution vs. intelligent design in program patching. Technical Report https://escholarship.org/uc/item/3z8926ks, UC Davis: College of Engineering, 2013.

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation*, pages 209–224, San Diego, CA, USA, 2008.

[19] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with Jolt. In *European Conference on Object Oriented Programming*, pages 609–633, 2011.

[20] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *International Conference on Sofware Engineering*, pages 782–791, 2013.

[21] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *International Symposium on Foundations of Software Engineering*, pages 237–246, 2010.

[22] Zack Coker and Munawar Hafiz. Program transformations to fix C integers. In *International Conference on Sofware Engineering*, pages 792–801, 2013.

[23] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Automated Software Engineering*, pages 550–554, 2009.

[24] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Automated Software Engineering*, pages 433–436, 2007.

[25] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, pages 65–74, 2010.

[26] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis*, pages 233–244, 2006.

[27] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

[28] Thomas Durieux, Matias Martinez, Martin Monperrus, Romain Sommerard, and Jifeng Xuan. Automatic repair of real bugs: An experience report on the Defects4J dataset. *CoRR*, abs/1505.07002, 2015.

[29] How science goes wrong. *Economist*, 409(8858), October 2013.

[30] Uwe Egly and Carsten Sinz, editors. *17th International Conference on the Theory and Applications of Satisfiability Testing (SAT)*, July 2014.

[31] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: A tool for repairing complex data structures. In *International Conference on Software Engineering*, pages 855–858, 2008.

[32] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference*, pages 965–972, 2010.

[33] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, pages 177–187, 2012.

[34] Zachary P. Fry and Westley Weimer. A human study of fault localization accuracy. In *International Conference on Software Maintenance*, pages 1–10, 2010.

[35] Divya Gopinath, Sarfraz Khurshid, Diptikalyan Saha, and Satish Chandra. Data-guided repair of selection statements. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 243–253, 2014.

[36] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188, 2011.

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

[38] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In *Fundamental Approaches to Software Engineering*, pages 267–280, 2004.

[39] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and control flow-based test adequacy criteria. In *International Conference on Software Engineering*, pages 191–200, 1994.

[40] Matti Järvisalo and Allen Van Gelder, editors. *16th International Conference on the Theory and Applications of Satisfiability Testing (SAT)*, July 2013.

[41] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. BugFix: A learning-based tool to assist developers in fixing bugs. In *International Conference on Program Comprehension*, 2009.

[42] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, 2010.

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

[44] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, pages 389–400, 2011.

[45] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Foundations of Software Engineering*, FSE 2014, pages 654–665, 2014.

[46] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, pages 266–276, 2014.

[47] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Sofware Engineering*, pages 802–811, 2013.

[48] John C. Knight and Paul Ammann. An experimental evaluation of simple methods for seeding program errors. In *International Conference on Software Engineering*, pages 337–342, 1985.

[49] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, pages 3–13, 2012.

[50] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Representations and operators for improving evolutionary software repair. In *Genetic and Evoluationary Computation Conference*, pages 959–966, 2012.

[51] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[52] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[53] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, pages 15–26, 2005.

[54] Peng Liu and Charles Zhang. Axis: Automatically fixing atomicity violations through solving control constraints. In *International Conference on Software Engineering*, pages 299–309, 2012.

[55] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[56] Martin Monperrus. A critical review of "Automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *International Conference on Software Engineering*, pages 234–242, 2014.

[57] National Institute of Standards and Technology (NIST). Samate — software software assurance metrics and tool evaluation, 2012.

[58] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Sofware Engineering*, pages 772–781, 2013.

[59] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *Transactions on Evolutionary Computation*, 15(2):166–192, 2011.

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

[61] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449, 2014.

[62] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. *SIGPLAN Not.*, 49(6):408–418, June 2014.

[63] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–102, Big Sky, MT, USA, October 12–14, 2009.

[64] President's council of advisors on science and technology (PCAST) public meeting agenda. http://www.whitehouse.gov/sites/default/files/microsites/ostp/PCAST/pcast_public_agenda_jan_2014.pdf.

[65] Programming language popularity. http://langpop.com.

[66] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *International Conference on Software Maintenance (ICSM)*, pages 180–189, Eindhoven, The Netherlands, September 2013.

[67] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, pages 191–201, 2013.

[68] Eric Schulte, Zachary Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, To Appear, 2013.

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

[70] Susan Elliot Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *International Conference on Software Engineering*, pages 74–83, 2003.

[71] Edward K. Smith, Earl Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Bergamo, Italy, September 2015.

[72] SPEC open systems group policies and procedures document, August 2013.

[73] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[74] Westley Weimer. Advances in automated program repair and a call to arms. In *The 5th International Symposium on Search Based Software Engineering (SSBSE)*, pages 1–3, St. Petersburg, Russia, August 2013.

[75] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE)*, pages 356–366, Novemvber 2013.

[76] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

[77] Wikipedia. Python (programming language). http://en.wikipedia.org/wiki/Python_(programming_language), February 2014.

[78] Josh L. Wilkerson, Daniel R. Tauritz, and James M. Bridges. Multi-objective coevolutionary automated software correction. In *Genetic and Evolutionary Computation Conference*, pages 1229–1236, 2012.

[79] Computer World. http://www.computerworld.com/s/article/9179538/Google_calls_raises_Mozilla_s_bug_bounty_for_Chrome_flaws, Accessed Feb, 2014.

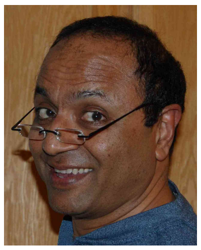
[80] Sai Zhang, Darioush Jalalinasab, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 2014.

Claire Le Goues received the BA degree in computer science from Harvard University and the MS and PhD degrees from the University of Virginia. She is an assistant professor in the School of Computer Science at Carnegie Mellon University, where she is primarily affiliated with the Institute for Software Research. She is interested in how to construct high-quality systems in the face of continuous software evolution, with a particular interest in automatic error repair. More information is available at: http://www.cs.cmu.edu/~clegoues.

Neal Holtschulte received the BA degree in mathematics from Williams College. He is a PhD student at the University of New Mexico, where he studies automated program repair and biologically inspired computation with Professor Melanie Moses. He is also interested in genetic programming, program structure, and computer science education.

Edward (Ted) K. Smith is a PhD student in the College of Information and Computer Science at the University of Massachusetts, Amherst. He received the BS degree from the University of Maryland in 2013. His research interests include human factors, programming languages, and software engineering. More information is available on his homepage: https://people.cs.umass.edu/tedks/.

Yuriy Brun is an Assistant Professor in the College of Information and Computer Science at the University of Massachusetts, Amherst. He received the PhD degree from the University of Southern California in 2008 and the MEng degree from the Massachusetts Institute of Technology in 2003. He completed his postdoctoral work in 2012 at the University of Washington, as a CI Fellow. His research focuses on software engineering, distributed systems, and self-adaptation. He received an NSF CAREER award in 2015, a Microsoft Research Software Engineering Innovation Foundation Award in 2014, and an IEEE TCSC Young Achiever in Scalable Computing Award in 2013. He is a member of the IEEE, the ACM, and ACM SIGSOFT. More information is available on his homepage: http://www.cs.umass.edu/~brun/.



Premkumar Devanbu received his B.Tech from IIT Madras, in Chennai, India, and his PhD from Rutgers University. After spending nearly 20 years as both a developer and a researcher at Bell Labs and its various offshoots, he left Industry to join the CS faculty at UC Davis in late 1997, where he is now Professor of Computer Science.

Stephanie Forrest received the BA degree from St. John's College and the MS and PhD degrees from the University of Michigan. She is currently Regents Distinguished Professor of Computer Science at the University of New Mexico and a member of the External Faculty of the Santa Fe Institute. Her research studies complex adaptive systems, including immunology, evolutionary computation, biological modeling, and computer security. She is an IEEE fellow.

Westley Weimer received the BA degree in computer science and mathematics from Cornell University and the MS and PhD degrees from the University of California, Berkeley. He is currently an associate professor at the University of Virginia. His main research interests include static and dynamic analyses to improve software quality and fix defects.