Clarifications on the Construction and Use of the ManyBugs Benchmark

Claire Le Goues, Yuriy Brun, *Senior Member, IEEE*, Stephanie Forrest, *Fellow, IEEE*, and Westley Weimer

HIGH-QUALITY research requires timely dissemination and the incorporation of feedback. Since the publication of the ManyBugs benchmark [2] and its release on http://repairbenchmarks.cs.umass.edu/, Fan Long and Martin Rinard have provided feedback on the benchmark's construction and use. Here, we describe that feedback and our subsequent improvements to the ManyBugs benchmark. We thank these researchers for the feedback.

php Test Harness. The **php** subject is a language interpreter. Both its testing framework and the tests themselves are effectively written in **php** (the tests are actually written in a variant of **php** called phpt). To run a test, a **php** interpreter (the *testing* interpreter) runs the testing framework, which loads a second **php** interpreter (the *tested* interpreter) to interpret the test **php** program. The test framework (again, executed by the *testing* interpreter) captures the output of the *tested* interpreter and compares it to the expected output.

Automated repair techniques produce variant **php** interpreters, which should naturally serve as the *tested* interpreters. However, the answer to the question of what should serve as the *testing* interpreter is less obvious. **php**'s default test harness configuration uses the same version of the interpreter for both the *tested* and *testing* interpreter. However, **php** may be configured via a command-line argument to use a different interpreter, such as the unmodified defective version, or a separate, manually-repaired version. Testing results—whether a test passes or fails—can differ based on which version is used as the *testing* interpreter. For example, theoretically, automated repair could produce a variant that, when interpreting the testing framework's code that compares a test's output to the expected output, always deems the results identical. Such a variant would produce different test results than would a different, unmodified version of the interpreter.

Absent an established answer on how to augment or improve the developer-provided test framework with automatic repair in mind, we initially configured the **php** scenarios to follow the developer-provided defaults. Thus, our reported experimental results (Section 6.2 in [2]) use the variant **php** interpreter as both *tested* and *testing* interpreter. Here, Fig. 1 augments Fig. 4 in [2] to include the repair results on the **php** scenarios using an unmodified version of **php** (version 5.3.1) as the *testing* interpreter. Row "**php** default" is identical to that in [2]; row "**php** alternative" shows the newly-computed results. We kept the experimental settings the same between the runs for consistency with the previous work, with one exception: The new AE experiments are run *without* the "super mutant" optimization (which typically improves efficiency).

We have added the repair logs and other data associated with these experiments to the ManyBugs release (at http://repairbenchmarks.cs.umass.edu/). We have also added support files and instructions, appending to the ManyBugs README, to support their reproduction on the previously-released AWS virtual machines.

Guarantees. One of the requirements of the ManyBugs defect scenarios was that "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" (Section 3.1 in [2]). We verified that the specification cannot be satisfied by the following programs:

(1) `/bin/false`
(2) `main(){ while(1) { }}`
(3) `main(){ /*immediate segmentation fault */}`
(4) an empty module.

Note that not all degenerate programs are caught by test output: timeouts on test execution catch infinite loops, and many scenarios do not compile or link trivial modules. Different approaches to constructing or configuring degenerate test programs may find different results.

libtiff Test Suite Quality. The test suites used in the ManyBugs benchmark are written by the respective project developers, and thus vary in quality. In this vein, it is worth noting that a number of the **libtiff** test cases in particular check only for a non-erroneous program exit code, and nothing else. Accordingly, when using the benchmark, researchers should be aware that these particular tests may not indicate program correctness beyond the program exit code.

*Discussion.*The above clarifications primarily concern the test suites provided with the ManyBugs scenario programs. All project test suites in ManyBugs are written by the respective projects' developers, and thus vary in quality; This is to be expected in real-world test suites [6]. However, this also means that the strength of the guarantees that such test suites provide about automatically-generated patches can vary. Moreover, different choices for test suite configuration and construction can produce different results when used to assess different patch generation techniques.

Assessing patch quality independently of tested behavior or otherwise guarding against degenerate patches that make tests pass without properly repairing the broken functionality is an unsolved research problem, and the subject of ongoing work [5], [7]. Absent a definitive understanding of how best to use developer-provided test suites to assess candidate repair quality, and to avoid interposing our own biases in benchmark construction, we sought to use these test suites in as unmodified a manner as possible. We are heartened by recent work that focuses on understanding and quantifying the quality problem, as well as on techniques that produce higher-quality patches, e.g., [1], [3], [4].

We look forward to ongoing discussion of these and related issues, with a goal of advancing collective scientific understanding of (1) what it means for a patch to be of high-quality, (2) how to measure that quality by using, augmenting, or side-stepping altogether the developer-provided test suites, and (3) how to automatically produce high-quality patches, improving software quality while decreasing the cost of developing it.

- *C. Le Goues is with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: clegoues@cs.cmu.edu.*
- *Y. Brun is with the College of Information and Computer Science, University of Massachusetts at Amherst, Amherst, MA 01003. E-mail: brun@cs.umass.edu.*
- *S. Forrest is with the Department of Computer Science, University of New Mexico, Albuquerque, NM 87131. E-mail: forrest@cs.unm.edu.*
- *W. Weimer is with the Computer Science and Engineering, University of Michigan, Ann Arbor, MI 48109. E-mail: weimerw@umich.edu.*

Manuscript received 13 Aug. 2017; accepted 18 Sept. 2017. Date of publication 21 Sept. 2017; date of current version 20 Nov. 2017.
(Corresponding author: Claire Le Goues.)
Recommended for acceptance by M. Dwyer.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2755651

Program	GenProg Defects repaired	GenProg Time (min)	GenProg Fitness evals	TrpAutoRepair Defects repaired	TrpAutoRepair Time (min)	TrpAutoRepair Fitness evals	AE Defects repaired	AE Time (min)	AE Fitness evals
`php` default	54/104	181	5.2	56/104	180	1.1	53/104	441	1.1
`php` alternative	19/104	210	6.1	27/104	193	1.2	17/104	471	1.0

Fig. 1. ManyBugs: Baseline results of running GenProg v2.2, TrpAutoRepair, and AE v3.0 on the 104 `php` defects from the ManyBugs benchmark using the variant `php` interpreter to interpret both the testing framework and the test (`php` default) and an unrepaired `php` interpreter to interpret the testing framework and the variant `php` interpreter to interpret the test (`php` alternative). For each of the repair techniques, we report the number of defects repaired; 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 and reproduction instructions, are available for download with the dataset.

Acknowledgments

This work is supported by the US National Science Foundation under grants no. CCF-1563797 and CCF-1564162.

References

[1] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2015, pp. 295–306. doi: 10.1109/ASE.2015.60.

[2] C. L. Goues, et al., "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Trans. Softw. Eng.*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015. doi: 10.1109/TSE.2015.2454513.

[3] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2016, pp. 298–312. doi: 10.1145/2837614.2837617.

[4] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. Int. Conf. Softw. Eng.*, May 2016, pp. 691–701. doi: 10.1145/2884781.2884807.

[5] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36. doi: 10.1145/2771783.2771791.

[6] Research Triangle Institute, "The economic impacts of inadequate infrastructure for software testing," *NIST Planning Report 02–3, Acquisition Assistance Division*, National Inst. Standards Technol. Gaithersburg, MD, USA, RTI Project Number 7007.011, May 2002.

[7] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, Sep. 2015, pp. 532–543. doi: 10.1145/2786805.2786825.