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 phpct). 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 interpreters. 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 an unmodified 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.
**ACKNOWLEDGMENTS**

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

**REFERENCES**


### Table 1

<table>
<thead>
<tr>
<th>Program</th>
<th>GenProg</th>
<th>TrpAutoRepair</th>
<th>AE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Defects</td>
<td>Time</td>
<td>Defects</td>
</tr>
<tr>
<td></td>
<td>repaired</td>
<td>(min)</td>
<td>repaired</td>
</tr>
<tr>
<td>php default</td>
<td>54/104</td>
<td>181</td>
<td>56/104</td>
</tr>
<tr>
<td>php alternative</td>
<td>19/104</td>
<td>210</td>
<td>27/104</td>
</tr>
</tbody>
</table>

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.