2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Poster: How Developers Debug Software The DBGBENCH Dataset

Marcel Böhme* Ezekiel O. Soremekun† Sudipta Chattopadhyay‡ Emamurho Ugherughe† Andreas Zeller†

*School of Computing, National University of Singapore, Singapore, marcel.boehme@acm.org
†CISPA, Saarland University, Germany, {soremekun@cs,s9emughe@stud,zeller@cs}.uni-saarland.de
‡Singapore University of Technology and Design, Singapore, sudipta_chattopadhyay@sutd.edu.sg

***Abstract*—How do professional software engineers debug computer programs? In an experiment with 27 real bugs that existed in several widely used programs, we invited 12 professional software engineers, who together spent one month on localizing, explaining, and fixing these bugs. This did not only allow us to study the various tools and strategies used to debug the same set of errors. We could also determine exactly which statements *a developer* would localize as faults, how *a developer* would diagnose and explain an error, and how *a developer* would fix an error – all of which software engineering researchers seek to automate. Until now, it has been difficult to evaluate the effectiveness and utility of automated debugging techniques without a user study. We publish the collected data, called DBGBENCH, to facilitate the effective evaluation of automated fault localization, diagnosis, and repair techniques w.r.t. the judgement of human experts.**

***Keywords*-Debugging in Practice, Fault Localization, Bug Diagnosis, Software Repair, User as Tool Benchmark, Tool Evaluation**

I. INTRODUCTION

In the software engineering community, the past decade has seen a surge in automated debugging techniques designed to assist programmers in localizing, explaining, and fixing faults in software. The bulk of automated debugging research is on Automated Fault Localization (AFL), that is, techniques that produce a ranked list of suspicious locations in the program. A recent survey cites more than 400 publications on AFL [1]. This is in stark contrast to the recent [2] (and antique [3]) finding that most practitioners have *never* used an AFL tool. Parnin and Orso [4] set out to shed light on this dichotomy and conducted a small user study with an AFL tool. Even after inspecting *the* faulty statement in the list of suspicious statements (whether coincidently or not), 9 of 10 participants would spend another 10 minutes, on average, before stopping to provide a diagnosis. Even when the faulty statement was ranked artificially high, developers did not consider the AFL tool more effective than traditional debugging.

Do we really understand how our tools can address the real-world debugging needs of software engineering professionals at work? Actually, not only do we know very little about how practitioners debug; we also lack data and methods that would allow us to check novel tools against practitioners' needs. Given an error, which locations would a human expert localize as faulty, how would she explain the chain of events leading to the error, and how would she fix it?

The best means of evaluating novel automated debugging tools is by way of *user studies*. However, user studies are expensive and take much time. In fact, our own user study cost several thousand US dollars and two years of designing the experiment, constructing a remote infrastructure, conducting sandbox and pilot studies, and recruiting software engineering professionals. These professionals together spent almost one month with bug diagnosis and repair. This puts our experiment perhaps among the *longest-running user studies* in the history of automated debugging. Between 1981 and 2011, Parnin and Orso [4] could identify no more than a handful of articles that discussed a user study involving software professionals. Just one study [5] also involved real errors in a real program. Perhaps we can reduce the cost of user studies by involving students? While there is some experience in favor [6], most colleagues warn that students *cannot* be representative for experienced software engineering professionals [7], [8], [9], [10]. In fact, our own evidence is against students as participants. The pilot study of our experiment was conducted with five students. On average, a student fixed one (1) error in eight (8) hours while a professional fixed 27 errors in 21.5 hours.

To allow the effective evaluation of automated debugging techniques without the cost and effort involved in user studies, we collected data in the following experiment with actual practitioners, and publish this as the DBGBENCH dataset.

II. EXPERIMENT

We used the 27 real bugs from COREBENCH [11] which were systematically extracted from the 10,000 most recent commits and the associated bug reports. We asked 12 software engineering professionals from 6 countries to debug these software errors. For each error, they got a small but succinct bug report, the buggy source code & executable, and a test case that fails because of this error. We asked the developers to point out the buggy program statements (*fault localization*), explain how the error comes about (*bug diagnosis*), and to develop a patch (*bug fixing*). From this data, we constructed DBGBENCH, a benchmark to evaluate automated debugging techniques w.r.t. human expert judgement. An example of the benchmark is shown in Figure 1. Apart from the dataset, we publish the protocol, questionnaires, and full setup on our webpage, and encourage fellow researchers to repeat our experiment for other programs and programming languages.

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.94

Find "-mtime [+-n]" is broken (behaves as "-mtime n")

```
Lets say we created 1 file each day in the last 3 days:
$ mkdir tmp
$ touch tmp/a -t $(date --date="yesterday" +"%y%m%d%H%M")
$ touch tmp/b -t $(date --date="2 days ago" +"%y%m%d%H%M")
$ touch tmp/c -t $(date --date="3 days ago" +"%y%m%d%H%M")

Running a search for files younger than 2 days, we expect
$ ./find tmp -mtime -2
tmp
tmp/a

However, with the current grep-version, I get
$ ./find tmp -mtime -2
tmp/b

Results are the same if I replace -n with +n, or just n.
```

(a) Bug Report and Test Case

If `find` is set to print files that are strictly younger than n days (-mtime -n), it will instead print files that are exactly n days old. The function get_comp_type actually increments the argument pointer timearg (`parser.c:3175`). So, when the function is called the first time (`parser.c:3109`), timearg still points to '-'. However, when it is called the second time (`parser.c:3038`), timearg already points to 'n' such that it is incorrectly classified as COMP_EQ (`parser.c:3178`; exactly n days).

(b) Bug diagnosis and Fault Locations

Example Correct Patches

- Copy timearg and restore after first call to get_comp_type.
- Pass a copy of timearg into first call of get_comp_type.
- Pass a copy of timearg into call of get_relative_timestamp.
- Decrement timearg after the first call to get_comp_type.

Example an Incorrect Patch

- Restore timearg only if classified as COMP_LT (*Incomplete Fix* because it does not solve the problem for -mtime +n).

(c) Examples of (in-)correct Patches

Fig. 1. DBGBENCH Example: For the error `find.66c536bb`, we show (a) the bug report and test case that a participant receives to reproduce the error, (b) the bug diagnosis that the participants provide (incl. fault locations), and (c) examples of ways how to patch the error (in-)correctly.

A. Automating the Diagnosis of Software Bugs

We find the very first evidence that bug diagnosis is indeed no subjective matter. Most participants provide essentially the same explanation for an error. *Suppose, everyone provided a different explanation or blamed different locations as the root cause of an error: How could there ever be consensus about the effectiveness or utility of an auto-generated bug diagnosis?* For each error, DBGBENCH provides an English explanation of the chain of events that lead up to the error (Fig. 1-b). This explanation is in agreement with 10 out of 12 participants, on average. However, while agreeable participants were often very confident about the correctness of their diagnosis the disagreeable participants were only slightly confident, providing further evidence in favor of our aggregated diagnoses.

Automated Fault Localization (AFL), the identification of a ranked list of most suspicious statements, is a major topic in automated debugging research; a recent literature survey on AFL cites more than 400 papers [1]. A common measure of AFL effectiveness is the proportion of suspicious locations that a developer would need to examine before *the* faulty location is found. However, we observe that there is no single fault location that dominates a bug diagnosis, neither semantically nor syntactically. Often, it is the complex interaction among several statements that bring about an error. This might explain the negative results of Parnin and Orso [4]. The middle 50% of bug diagnoses reference between three and four continuous code regions that can be distributed over several files. Interestingly, the produced software *patches* tend to be local to one function. Still, there was an overlap between the statements mentioned in the diagnosis and the statements changed in the patch only for 69% of submitted patches. For a memory leak, for instance, the diagnosis references the statements causing the leak while the patch releases the memory potentially anywhere in the program. *This motivates further research in auto-generated patches as aids to bug diagnosis* (e.g., [12]).

Our participants perceived four of 27 bugs as very difficult to diagnose. Asked to provide a rationale, our participants told us that certain flags, functions, or data structures were left undocumented. This impeded program comprehension, a prerequisite for effective debugging and *motivates further research in automated code documentation*.

B. Automating the Repair of Software Bugs

Even professional software engineers with at least seven years experience, each of whom spent more then two days debugging 27 bugs in only two programs, submit *plausible but incorrect patches*. While 282 out of 291 of submitted patches pass the previously failing test case, only 170 patches (58%) are actually correct in the sense that they also pass our code review. For each error, DBGBENCH provides high-level examples of correct and incorrect patches (Fig. 1-c). For incorrect patches, we provide a rationale as to why we classify them as incorrect. It is interesting to note that the *principal causes of patch incorrectness could be addressed by automated regression testing* techniques (e.g., [13], [14]): 124 patches are incorrect because they either introduce new bugs or they did not fix the bug completely. 34 patches are incorrect because they treat the symptom rather than the root cause, for instance, by deleting the failing assertion. The remaining ten incorrect patches can be classified as incorrect workarounds. *This motivates research in combining automated regression test generation and software repair to increase the correctness of (auto-generated) software patches.*

In terms of fix ingredients, we found that *one-third of patches exclusively affect the control-flow*. Such patches may be efficiently generated by automated repair techniques such as SPR [15]. Only very few patches would require the synthesis of complex functions. However, many patches actually add new statements, like a function call to release resources. Most patches could not be generated with simple mutation operators. Yet, three of the four bugs that were perceived to be very difficult to diagnose are actually caused by a simple operator fault and, hence, perceived to be easy to fix.

III. DBGBENCH DATASET

We publish the full dataset of DBGBENCH together with the protocol, questionnaires, setup, and virtual infrastructure on our webpage. We encourage fellow researchers to repeat our experiment for other programs and programming languages and utilize DBGBENCH to evaluate their novel automated fault localization, bug diagnosis, and software repair techniques with respect to human expert judgement.

- http://www.st.cs.uni-saarland.de/debugging/dbgbench/

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[2] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld, "Studying the advancement in debugging practice of professional software developers," *Software Quality Journal*, pp. 1–28, 2016.

[3] H. Lieberman, "The debugging scandal and what to do about it (introduction to the special section)," *Communications of the ACM*, vol. 40, no. 4, pp. 26–29, 1997.

[4] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 20th International Symposium on Software Testing and Analysis*, ser. ISSTA, 2011, pp. 199–209.

[5] A. J. Ko and B. A. Myers, "Finding causes of program output with the Java Whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '09, 2009, pp. 1569–1578.

[6] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, pp. 666–676.

[7] A. J. Ko, T. D. Latoza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, pp. 110–141, Feb. 2015.

[8] M. D. Penta, R. E. K. Stirewalt, and E. Kraemer, "Designing your next empirical study on program comprehension," in *15th IEEE International Conference on Program Comprehension (ICPC '07)*, June 2007, pp. 281–285.

[9] G. J. Myers, "A controlled experiment in program testing and code walkthroughs/inspections," *Communications of the ACM*, vol. 21, no. 9, pp. 760–768, Sep. 1978.

[10] B. Curtis, "By the way, did anyone study any real programmers?" in *Proceedings of the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, 1986, pp. 256–262.

[11] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, 2014, pp. 105–115.

[12] Y. Tao, J. Kim, S. Kim, and C. Xu, "Automatically generated patches as debugging aids: A human study," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 64–74.

[13] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Regression tests to expose change interaction errors," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 334–344.

[14] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Partition-based regression verification," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 302–311.

[15] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 166–178.