Where Is the Bug and How Is It Fixed? An Experiment with Practitioners*

Marcel Böhme
National University of Singapore, Singapore
marcel.boehme@acm.org

Ezekiel O. Soremekun
Saarland University, Germany
soremekun@cs.uni-saarland.de

Sudipta Chattopadhyay
Singapore University of Technology and Design, Singapore
sudipta_chattopadhyay@sutd.edu.sg

Emamurho Ugherughe
SAP Berlin, Germany
emamurho@gmail.com

Andreas Zeller
Saarland University, Germany
zeller@cs.uni-saarland.de

ABSTRACT

Research has produced many approaches to automatically locate, explain, and repair software bugs. But do these approaches relate to the way practitioners actually locate, understand, and fix bugs? To help answer this question, we have collected a dataset named DbgBench—the correct fault locations, bug diagnoses, and software patches of 27 real errors in open-source C projects that were consolidated from hundreds of debugging sessions of professional software engineers. Moreover, we shed light on the entire debugging process, from constructing a hypothesis to submitting a patch, and how debugging time, difficulty, and strategies vary across practitioners and types of errors. Most notably, DbgBench can serve as *reality check* for novel automated debugging and repair techniques.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging;

KEYWORDS

Debugging in practice, user as tool benchmark, evaluation, user study

ACM Reference Format:
Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where Is the Bug and How Is It Fixed? An Experiment with Practitioners. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8, 2017 (ESEC/FSE'17),* 12 pages.
https://doi.org/10.1145/3106237.3106255

1 INTRODUCTION

In the past decade, research has produced a multitude of automated approaches for fault localization, debugging, and repair. Several benchmarks have become available for the *empirical evaluation* of such approaches. For instance, CoREBench [7] and Defects4J [13] contain a large number of real errors for C and Java, together with developer-provided test suites and bugfixes. Using such benchmarks, researchers can make *empirical claims* about the efficacy of their tools and techniques. For instance, an effective fault localization technique would rank very high a statement that was changed in the bugfix [49]. The assumption is that practitioners would identify the same statement as *the* fault. An effective auto-generated bugfix would pass all test cases [27]. The assumption is that practitioners would accept such fixes. Unfortunately, debugging is not that simple, particularly not for humans. In this paper, we provide another kind of benchmark; one that allows *reality checks*.

Given the complexity of the debugging process, one might assume that it would be standard practice to evaluate novel techniques by means of user studies [24]: Does the tool fit into the process? Does it provide value? How? Yet, how humans actually debug is still not really well explored. Between 1981 and 2010, Parnin and Orso [31] identified only a handful of articles that presented the results of a user study—none of which involved actual practitioners *and* real errors. Since the end of 2010, we could identify only *three (3) papers* that evaluated new debugging approaches with actual practitioners and real errors [8, 15, 41].

In this paper, we do not attempt to evaluate a specific approach. Instead, we shed light on the *entire debugging process*. Specifically, we investigate how debugging time, difficulty, and strategies vary across practitioners and types of errors. For our benchmark, we elicit which fault locations, explanations, and patches *practitioners* produce. We used 27 real bugs from CoREBench [7] 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:

Participants *received* for each error

- a small but succinct bug report,
- the buggy source code and executable, and
- a test case that fails because of this error.

We *asked* participants

- to point out the buggy statements (*fault localization*),
- to explain how the error comes about (*bug diagnosis*), and
- to develop a patch (*bug fixing*).

We *recorded* for each error

- their *confidence* in the correctness of their diagnosis / patch,
- the *steps* taken, the *tools* and *strategies* used, and
- the *time* taken and *difficulty* perceived in both tasks.

*All authors conducted this work while affiliated with Saarland University, Germany.

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106255

(a) Bug Report and Test Case

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.
```

(b) Bug diagnosis and Fault Locations

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).

(c) Examples of (in-)correct Patches

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*).

Figure 1: An excerpt of DbgBench. 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 we consolidated from those provided by participants (including fault locations), and (c) examples of ways how participants patched the error correctly or incorrectly.

We *analyzed* this data and

- derived for each error important fault locations and a diagnosis
- evaluated the correctness of each submitted patch, and
- provide new test cases that fail for incorrect patches.

Findings. To the best of our knowledge, we find the first evidence that debugging *can* actually be automated and is no subjective endeavour. In our experiment, *different* practitioners provide essentially the *same* fault locations and the *same* bug diagnosis for the *same* error. If humans disagreed, how could a machine ever produce the "correct" fault locations, or the "correct" bug diagnosis? Moreover, we find that many of the participant-submitted patches are actually incorrect: While 97% of all patches are *plausible*, i.e., pass the failing test case, only 63% are *correct*, i.e., pass our code review. Taking human error out of the equation provides *opportunities for automated program repair* [30]. We also find that three in four incorrect patches introduce regression errors or do not fix the error completely. This provides *opportunities for automated regression testing* [5, 6]. We also find that practitioners are wary of debugging automation. They might quickly adopt an auto-repair tool for crashes but seem reluctant for functional bugs. Actual tools might prove such beliefs unwarranted.

Benchmark. Since participants agree on essential bug features, it is fair to treat their findings as *ground truth*. We have compiled our study data for all 27 bugs into a *benchmark* named DbgBench [43], which is the second central contribution of this paper. An excerpt of DbgBench for a specific bug is shown in Figure 1. DbgBench can be used in *cost-effective user studies* to investigate how debugging time, debugging difficulty, and patch correctness improve with the novel debugging/repair aid. DbgBench can be used to evaluate *without a user study* how well novel automated tools perform against professional software developers in the tasks of fault localization, debugging, and repair.

2 STUDY DESIGN

The study design discusses our recruitment strategy, the objects and infrastructure, and the variables that we modified and observed in our experiment. The goal of the study design is to ensure that the design is appropriate for the objectives of the study. We follow the canonical design for controlled experiments in software engineering with human participants as recommended by Ko et al. [17].

2.1 Research Questions

The main objective of the experiment is to construct a benchmark that allows to evaluate automated fault localization, bug diagnosis, and software repair techniques w.r.t. the judgment of actual professional software developers. We also study the various aspects of debugging in practice and opportunities to automate diagnosis and repair guided by the following *research questions*.

RQ.1 Time and Difficulty. Given an error, how much time do developers spend understanding and explaining the error, and how much time patching it? How difficult do they perceive the tasks of bug diagnosis and patch generation?

RQ.2 Fault Locations and Patches. Which statements do developers localize as faulty? How are the fault locations distributed across the program? How many of the provided patches are plausible? How many are correct?

RQ.3 Diagnosis Strategies. Which strategies do developers employ to understand the runtime actions leading to the error?

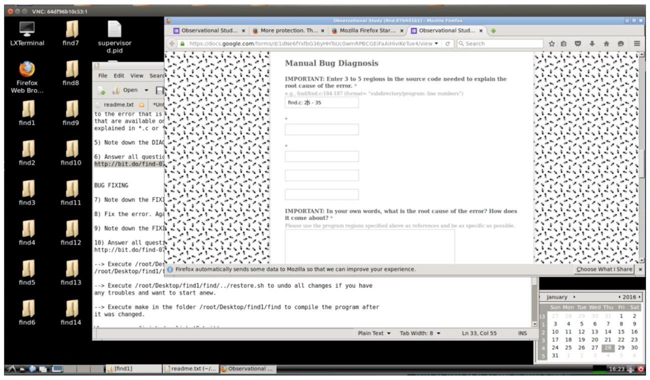
RQ.4 Repair Ingredients. What are the pertinent building blocks of a correct repair? How complex are the provided patches?

RQ.5 Debugging Automation. Is there a consensus among developers during fault localization and diagnosis? Hence, can debugging be automated? Do they believe that diagnosis or repair for an error will ever be automated and why?

2.2 Objects and Infrastructure

The *objects* under study are 27 real software errors systematically extracted from the bug reports and commit logs of two open-source C projects (find & grep). The *infrastructure* is a lightweight Docker container that can quickly be installed remotely on any host OS [44]. The errors, test cases, bug reports, source code, and the complete Docker infrastructure is available for download [43].

The objects originate from a larger error benchmark called CoREBench [7]. Errors were systematically extracted from the 10,000 most recent commits and the bug reports in four projects. Find and grep are well-known, well-maintained, and widely-deployed open-source C programs with a codebase of *17k and 19k LoC*, respectively. For each error, we provide a failing test case, a simplified bug report, and a large regression test suite (see Figure 1-a). We chose two subjects out of the four available to limit the time a participant spends in our study to a maximum of three working days and to help participants to get accustomed to at most two code bases.



Manual Bug Diagnosis

IMPORTANT: Enter 3 to 5 regions in the source code needed to explain the root cause of the error. *

e.g., find/find.c:184-187 (format= "subdirectory/program: line numbers")

find.c: 25 - 35

IMPORTANT: In your own words, what is the root cause of the error? How does it come about? *

Please use the program regions specified above as references and be as specific as possible.

Firefox automatically sends some data to Mozilla so that we can improve your experience.

```
to the error that is
that are available or
explained in *.c or

5) Note down the DIAC

6) Answer all questic
http://bit.do/find-07


BUG FIXING

7) Note down the FIXI

8) Fix the error. Aga

9) Note down the FIXI

10) Answer all questi
http://bit.do/find-07

--> Execute /root/Des
/root/Desktop/find1/f

--> Execute /root/Desktop/find1/find/../restore.sh to undo all changes if you have
any troubles and want to start anew.

--> Execute make in the folder /root/Desktop/find1/find to compile the program after
it was changed.
```

Figure 2: Screenshot of the provided virtual environment.

To conduct the study remotely and in an unsupervised manner, we developed a virtual environment based on Docker [43, 44]. The *virtual environment* is a lightweight Docker image with an Ubuntu 14.2 Guest OS containing a folder for each buggy version of either grep or find (27 in total). A script generates the *participant ID* for the responses by a participant. This ensures anonymity and prevents us from establishing the identity of any specific participant. At the same time we can anonymously attribute a response for a different error to the same participant to measure, for instance, how code familiarity increases over time. The same script does some *folder scrambling* to randomize the order in which participants debug the provided errors: The first error for one participant might be the last error for another. The scrambling controls for learning effects. For instance, if every participant would start with the same error, this error might incorrectly be classified as very difficult. The *docker image* contains the most common development and debugging tools for C, including gdb, vim, and Eclipse. Participants were encouraged to install their own tools and copy the created folders onto their own machine. A screenshot is shown in Figure 2.

2.3 Pilot Studies: Researchers and Students

Ko et al. [17] note that the design of a study with human participants is necessarily an iterative process. Therefore, a critical step in preparing an experiment is to run pilot studies. Hence, we first evaluated our design and infrastructure in a *sandbox pilot* where we, the researchers, were the participants. This allowed us to quickly catch the most obvious of problems at no extra cost. Thereupon, we sought to recruit several *graduate students* from Saarland University for the *pilot study*. We advertised the study in lectures, pasted posters on public bulletin boards, and sent emails to potentially interested students. From 10 interested students, we selected five (5) that consider their own level of skill in the programming with C as *advanced or experts*.[1] We conducted the pilot study as *supervised, observational study* in-house, in our computer lab. After filling the consent form and answering demographic questions, we introduced the errors and infrastructure in a small *hands-on tutorial* that lasted about 30 minutes. Then, the students had eight (8) hours, including a one hour lunch break, to debug as many errors as they could. We recorded the screen of each student using a *screen capturing* tool.

[1]Note that self-assessment of level of skill should always be taken with a grain of salt (cf. Dunning-Kruger effect [20]).

Independent of the outcome, all participants received EUR 50 as monetary compensation. While *none* of the data collected in the pilot studies was used for the final results, the pilot studies helped us to improve our study design in several ways:

1) **No Students.** For the main study, we would use only software engineering professionals. In seven hours our student participants submitted only *a sum total* of five patches. On average, a student fixed one (1) error in eight (8) hours (while in the main study a professional fixed 27 errors in 21.5 hours). The feedback from students was that they under-estimated the required effort and over-estimated their own level of skill.
2) **No Screen Capturing.** The video of only a single participant would take several Gigabyte of storage and it needs to be transferred online to a central storage. This was deemed not viable.
3) **Good Infrastructure.** The setup, infrastructure, and training material was deemed appropriate.

2.4 Main Study: Software Professionals

We make available the training material, the virtual infrastructure, the questionnaire that was provided for each error, the collected data [43]. The *experiment procedure* specifies the concrete steps a participant follows from the beginning of the experiment to its end.

Recruitment. First, participants would need to be recruited. To select our candidates, we designed an online questionnaire. The questionnaire asks general questions about debugging in practice after which developers have the option to sign up for the experiment. We sent the link to more than 2,000 developers on Github and posted the link to 20 software development usergroups at Meetup.com, on six (6) freelancer platforms, including Freelancer, Upwork, and Guru, and on social as well as professional networks, such as Facebook and LinkedIn. The job postings on the freelancer platforms were the most effective recruitment strategy. We started three advertisement campaigns in Aug'15, Mar'16, and July'16 following which we had the highest response rate lasting for about one month each. We received the first response in Aug'15 and the most recent response more than one year later in Oct'16. In total, we received 212 responses out of which 143 indicated an interest in participating in the experiment.

Selection. We selected and invited 89 professional software engineers based on their experience with C programming. However, in the two years of recruitment only 12 participants actually entered and completed the experiment. There are several reasons for the high attrition rate. Interested candidates changed their mind in the time until we sent out the invitation, when they understood the extent of the experiments (2–3 working days), or when they received the remote infrastructure and understood the difficulty of the experiment (17k + 19k unfamiliar lines of code).

Demographics. The final participants were *one researcher* and *eleven professional software engineers* from six countries (Russia, India, Slovenia, Spain, Canada, and Ukraine). All professionals had profiles with Upwork and at least 90% success rate in previous jobs.

Training. Before starting with the study, we asked participants to set up the Docker image and get familiar with the infrastructure. We made available *1 readme, 34 slides, and 10 tutorial videos* (~2.5 minutes each) that explain the goals of our study and provide details about subjects, infrastructure, and experimental procedure.

(1) **How difficult was it to understand the runtime actions leading to the error?**
(Not at all difficult, Slightly difficult, Moderately difficult, Very difficult, Extremely difficult)

(2) **Which tools did you use to understand the runtime actions leading to the error?**
[Textbox]

(3) **How much time did you spend understanding the runtime actions leading to the error?**
(1 minute or less, 2–5 minutes, 5–10 minutes, 10–20 minutes, ..., 50–60 minutes, 60 minutes or more)

(4) **Enter 3 to 5 code regions needed to explain the root cause of the error.**
[Textbox 1], [Textbox 2], [Textbox 3], [Textbox 4], [Textbox 5]

(5) **What is the root cause of the error? How does it come about?**
[Textbox]

(6) **How confident are you about the correctness of your explanation?**
(Not at all confident, Slightly confident, Moderately confident, Very confident, Extremely confident)

(7) **If you could not explain the error, what prevented you from doing so?**
[Textbox]

(8) **Which concrete steps did you take to understand the runtime actions?**
[Textbox]

(9) **Do you believe that the root cause of the error can be explained intuitively by the push of a button?**
Yes, in principle a tool might be able to explain this error. No, there will never be a tool that can explain this error.

(10) **Why do you (not) believe so?**
[Textbox]

Figure 3: Questions on the fault locations and bug diagnosis

(12) **How difficult was it to fix the error?**
(Not at all difficult, Slightly difficult, Moderately difficult, Very difficult, Extremely difficult)

(13) **How much time did you spend fixing the error?**
(1 minute or less, 2–5 minutes, 5–10 minutes, 10–20 minutes, ..., 50–60 minutes, 60 minutes or more)

(14) **IMPORTANT: Copy & paste the generated patch here.**
[Textbox]

(15) **In a few words and on a high level, what did you change to fix for the error?**
[Textbox]

(16) **How confident are you about the correctness of your fix?**
(Not at all confident, Slightly confident, Moderately confident, Very confident, Extremely confident)

(17) **In a few words, how did you make sure this is a good fix?**
[Textbox]

(18) **If you could not fix the bug, what prevented you from doing so?**
[Textbox]

(19) **Do you believe that this error can be fixed reliably by the push of a button?**
Yes, in principle a tool could fix this error reliably. No, there will never be a tool that can fix this error reliably.

(20) **Why do you (not) believe so?**
[Textbox]

Figure 4: Questions on generating the software patch

Participants could watch the slides and the tutorial videos at their own pace. The training materials are available [43]. Moreover, we informed them that they could contact us via Email in case of problems. We provided technical support whenever needed.

Tasks. After getting familiar with the infrastructure and the programs, participants chose a folder containing the first buggy version to debug. This folder contains a link to the questionnaire that they are supposed to fill in relation with the current buggy version. The text field containing the *participant's ID* is set automatically. The questionnaire contains the technical questions, is made available [43], and is discussed in Section 2.5 in more details. We asked each participant to spend approximately 45 minutes per error in order to remain within a 20 hour time frame. From the pilot study, we learned that incentive is important. So, we asked them to fix at least 80% of the errors in one project (e.g., grep) before being able to proceed to the next project (e.g., find).

Debriefing. After the experiment, participants were debriefed and compensated. We explained how the data is used and why our research is important. Participants would fill a final questionnaire to provide general feedback on experiment design and infrastructure. For instance, participants point out that sometimes it was difficult to properly distinguish time spent on diagnosis from time spent on fixing. Overall, the participants enjoyed the experiment and solving many small challenges in a limited time.

Compensation. It is always difficult to determine the appropriate amount for monetary compensation. Some guidelines [35] recommend the average hourly rate for professionals, the rationale being that professionals in that field cannot or will not participate without pay for work-time lost. Assuming 20 working hours and an hourly rate of USD 27, each participant received USD 540 in compensation for their time and efforts. The modalities were formally handled via Upwork [45].

2.5 Variables and Measures

The main objective of this study is to collect the fault locations, bug diagnoses, and software patches that each participant produced for each error. To assess the *reliability* of their responses, we use a triangulation question which asks for their *confidence* in the correctness of the produced artifacts. In addition to these artifacts, for each error, we also measure the *perceived difficulty* of each debugging task (i.e., bug diagnosis and bug fixing), the *time spent* with each debugging task, and their opinion on whether bug diagnosis or repair will ever be *fully automated* for the given error. The questions that we ask for each error are shown in figures 3 and 4.

To measure *qualitative attributes*, we utilize the 5-point Likert scale [29]. A 5-point Likert scale allows to measure otherwise qualitative properties on a symmetric scale where each item takes a value from 1 to 5 and the distance between each item is assumed to be equal.[2] For instance, for Question 12 in Figure 4, we can assign the value 1 to *Not at all difficult* up to the value 5 for *Extremely difficult*. An average difficulty of 4.7 would indicate that most respondents feel that this particular error is *very* to *extremely difficult* to fix while only few think it was *not at all difficult*.

We note that all data, including time, is self-reported rather than recorded during observation. Participants fill questionnaires and provide the data on their own. This allowed us to conduct the study fully remotely without supervision while they could work in their every-day environment. Since freelancers are typically payed by the hour, Upwork provides mechanisms to ensure that working time is correctly reported. While self-reports might be subject to cognitive bias, they also reduce observer-expectancy bias and experimenter bias [12]. Perry et al. [33] conducted an observational study with 13 software developers in four software development departments and found that the time diaries which were created by the developers *correspond sufficiently* with the time diaries that were created by observers. In other words, in the software development domain self-reports correspond sufficiently with independent researcher observations.

We checked the *plausibility* of the submitted patches by executing the complete test suite and the previously failing test case. We checked the *correctness* of the submitted patches using internal code reviews. Two researchers spent about two days discussing and reviewing the patches together. Moreover, we designate a patch as incorrect only if we can provide a rationale.

Generally, every qualitative analysis was conducted by at least one researcher and cross-checked by at least one other researcher.

3 STUDY RESULTS

Overall, *27 real errors* in 2 open-source C programs were diagnosed and patched by *12 participants* who together spent *29 working days*.

[2]However, the Likert-scale is robust to violations of the equal distance assumption: Even with larger distortions of perceived distances between items (e.g., slightly vs. moderately familiar), Likert performs close to where intervals are perceived equal [23].

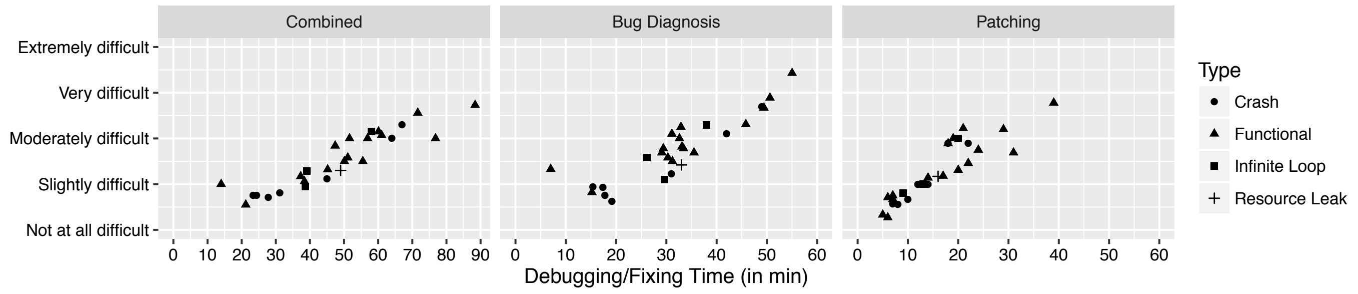


Figure 5: Relationship between average time spent and difficulty perceived for patching and diagnosing a bug. Each point is one of 27 bugs, the shape of which determines its bug type (i.e., crash, functional error, infinite loop, or resource leak).

RQ.1 Time and Difficulty

Our data on debugging time and difficulty can be used in cost-effective user-studies that set out to show how a novel debugging aid improves the debugging process in practice. We also elicit causes of difficulties during the manual debugging process.

Time and Difficulty. *On average, participants rated an error as moderately difficult to explain (2.8) and slightly difficult to patch (2.3). On average, participants spent 32 and 16 minutes on diagnosing and patching an error, respectively. There is a linear and positive relationship between perceived difficulty and the time spent debugging.* As we can see in Figure 5, participants perceived four errors to be *very difficult to diagnose.* These are three functional errors and one crash. Participants spent about 55 minutes diagnosing the error that was most difficult to diagnose. However, there are also nine errors perceived to be *slightly difficult to diagnose* with the main cluster situated between 15 and 20 minutes of diagnosis time. Participants perceived one (functional) error as *very difficult to patch* and spent about 40 minutes patching it. However, there are also two bugs perceived to be *not at all difficult to patch* and took about 5 minutes.

Why are some errors very difficult? There are four errors rated as *very difficult to diagnose.* In many cases, missing documentation for certain functions, flags, or data structures were mentioned as reasons for such difficulty. Other times, developers start out with an incorrect hypothesis before moving on to the correct one. For instance, the crash in grep.3c3bdace is caused by a corrupted heap, but the crash location is very distant from the location where the heap is corrupted. The crash and another functional error are caused by a simple operator fault. *Three of the four bugs* which are very difficult to diagnose are actually *fixed in a single line.* For the only error that is both very difficult to diagnose *and patch*, the developer-provided patch is actually very complex, involving 80 added and 30 deleted lines of code. Only one participant provided a correct patch.

RQ.2 Fault Locations and Patches

Our data on those program locations which practitioners need to explain how an error comes about (i.e., fault locations) can be used for a more *realistic evaluation* of automated fault localization, and motivates the development of techniques that point out *multiple* pertinent fault locations. Our data on multiple patches for the same error can be used to evaluate auto-repair techniques, and motivates research in automated repair and its integration with automated regression test generation to circumvent the considerable human error.

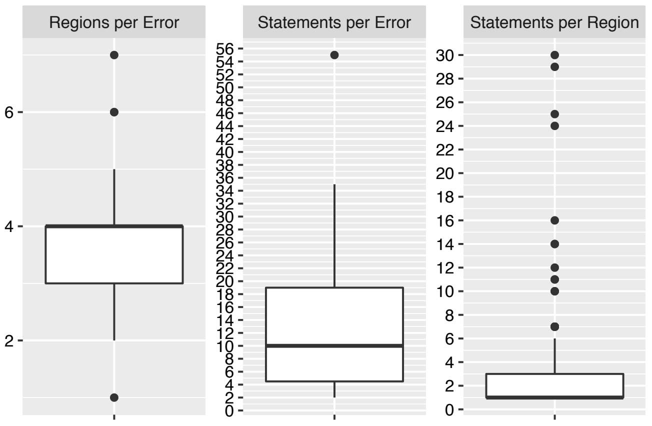


Figure 6: Boxplots showing the number of contiguous regions per bug diagnosis (left), the number of statements per diagnosis (middle), and the number of statements per contiguous region (right). For example, file.c:12-16,20 specifies six statements in two contiguous regions.

Fault Locations. *The middle 50% of consolidated bug diagnoses references three to four contiguous code regions, many of which can appear in different functions or files. In other words, practitioners often reference multiple statements to explain an error.* A *contiguous code region* is a consecutive sequence of program statements. In most cases, the regions for one error are localized in different functions and files. As shown in Figure 6, the majority of contiguous regions (below the median) contain only a single statement, the middle 50% contains between 1 and 3 statements. A typical bug diagnosis references 10 statements.

Patches. *While 282 out of 290 (97%) of the submitted patches are plausible and pass the provided test case, only 182 patches (63%) are actually correct and pass our code review.*[3] A *correct patch* does not introduce new errors and does not allow to provide other test cases that fail due to the same error. We determined *correctness* by code review and *plausibility* by executing the failing test case. For each incorrect patch, we also give a reason as to why it is incorrect and whether the test case passes.

[3]Note that participants were asked to ensure the plausibility of their submitted patch by passing the provided test case.

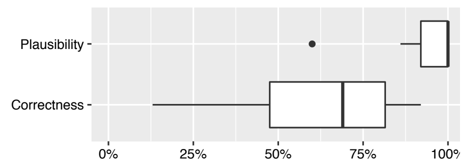


Figure 7: Boxplots showing the proportional patch correctness and plausibility. For instance, if the proportional patch correctness for an error is 50%, then half of the patches submitted for this error are correct. The boxplot shows the proportional plausibility and correctness *over all 27 errors.*

Figure 7 shows that the *median proportional plausibility* is 100%, meaning that for the majority of errors (above the median), all patches that participants submit pass the provided test case. Even for the middle 50% of errors, more than 90% of patches are plausible. However, the *median proportional correctness* is 69%, meaning that for the majority of errors (above the median), only 69% of patches submitted by participants pass our code review. For the middle 50% of errors only between 45% and 82% of patches are actually correct.

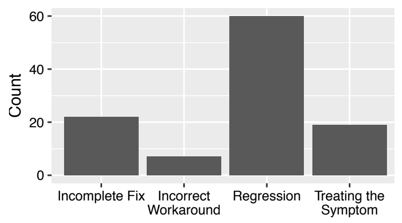


Figure 8: Histogram showing reasons why 108 of patches that were submitted by participants failed our code review.

Figure 8 shows that more than half of the incorrect patches (60 of 108) actually introduce regressions. A *regression* breaks existing functionality; we could provide a test that fails but passed before. 22 patches do not fix the error completely. An *incomplete fix* does not patch the error completely; we could provide a test that fails with and without the patch because of the bug. 19 patches are treating the symptom rather than fixing the error. A patch is *treating the symptom* if it does not address the root cause. For instance, it removes an assertion to stop it from failing. Seven (7) patches apply an incorrect workaround than fixing the error. An *incorrect workaround* changes an artifact that is not supposed to be changed, like a third-party library.

RQ.3 Bug Diagnosis Strategies

For each error, we asked participants which concrete steps they took to understand the runtime actions leading to the error. We analyzed 476 different responses for this question and we observed the following bug diagnosis strategies.

Classification. We extend the bug diagnosis strategies that have been identified by Romero and colleagues [14, 36]:

- (FR) *Forward Reasoning.* Programmers follow each computational step in the execution of the failing test.
- (BR) *Backward Reasoning.* Programmers start from the unexpected output following backwards to the origin.
- (CC) *Code Comprehension.* Programmers read the code to understand it and build a mental representation.
- (IM) *Input Manipulation.* Programmers construct a similar test case to compare the behavior and execution.
- (OA) *Offline analysis.* Programmers analyze an error trace or a coredump (e.g. via valgrind, strace).
- (IT) *Intuition.* Developer uses her experience from a previous patch.

Specifically, we identified the Input Manipulation (IM) bug diagnosis strategy. Developers would first modify the failing test case to construct a passing one. This gives insight into the circumstances required to observe the error. Next, they would compare the program states in both executions. IM is reminiscent of classic work on automated debugging [52], which might again reflect the potential lack of knowledge about automated techniques that have been available from the research community for over a decade.

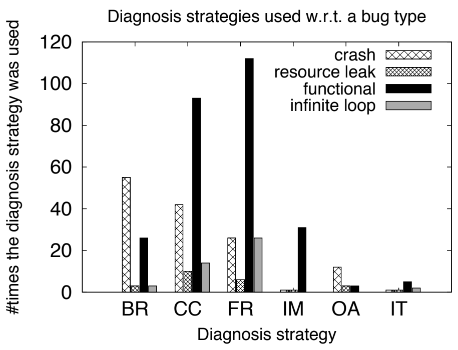


Figure 9: Diagnosis strategies for different error types.

Frequency. We discovered that *forward reasoning and code comprehension (FR+CC) are the most commonly used* diagnosis strategies in our study. The number of usage of different bug diagnosis strategies is shown in Figure 9 for the different error types. We observe that *past experience (IT) is used least frequently.* Many participants used input modification (IM) as diagnosis strategy. Therefore, the integration of automated techniques that implement IM (*e.g.* [52]) into mainstream debugger will help improve debugger productivity.

Error Type. We observe that forward reasoning (FR) is the most commonly used diagnosis strategy for bugs reflecting infinite loops (26 out of a total 45 responses). Intuitively, there is no last executed statement which can be used to reason backwards from. *Out of a total 137 responses for crash-related bugs, we found that backward reasoning (BR) was used 55 times.* Intuitively, the crash location is most often a good starting point to understand how the crash came about. For functional errors, 112 responses, out of a total 270 responses, reflect forward reasoning (FR). If the symptom is an unexpected output, the actual fault location can be very far from print statement responsible for the unexpected output. It may be better to start stepping from a location where the state is not infected, yet. Finally, we observed that input modification (IM) strategy was used for 31 out of 270 scenarios to diagnose functional errors. This was to understand what distinguishes the failing from a passing execution.

Tools. Every participant used a combination of trace-based and interactive debugging. For resource leaks, participants further used tools such as *valgrind* and *strace*. We also observed that participants use bug diagnosis techniques that have been automated previously [52], albeit with manual effort, to narrow down the pertinent sequence of events.

RQ.4 Repair Ingredients

Out of 290 submitted patches, 100 (34%) exclusively affect the control-flow, 87 (30%) exclusively affect the data-flow, while the remaining 103 patches (36%) affect both, control- and data-flow.

Control-Flow. In automated repair research, the patching of control-flow is considered tractable because of the significantly reduced search space [50]: Either a set of statements is executed or not. The frequency with which participants fix the control-flow may provide some insight about the effectiveness of such an approach for the errors provided with DbgBench. The control-flow is modified by 200 patches (69%). Specifically, a branch condition is changed by 126 patches and the loop or function flow is modified by 38 patches.[4] A new if-branch is added by 86 patches whereupon, in many cases, an existing statement is then moved into the new branch or a new function call is added.[5]

Data-Flow. The data-flow is modified by 187 of 290 submitted patches (64%). Specifically, 57 patches *modify* a variable value or function parameter. GenProg [27] copies, moves, or deletes existing program statements, effectively relying on the Plastic Surgery Hypothesis (PSH) [3]. In our study, the PSH seems to hold. 44 patches *move* existing statements while 29 patches *delete* existing statements. However, 73 patches *add* new variable assignments while 40 patches add new function calls, for instance to report an error or to release resources. A completely new variable is declared in 27 patches. Only 8 patches introduce complex functions that need to be synthesized.

Patch Complexity. On average, a submitted patch contained six (6) Changed Lines of Code (CLoC). The median patch contained 3 CLoC. The mean being to the far right of the median points to a skewed distribution. Indeed, there are many not very complex patches but there are a few that require more than 50 CLoC.

RQ.5 Debugging Automation

We investigate whether there is consensus among the developers during fault localization, diagnosis and fixing. Suppose, there is not. Then, how should there ever be consensus on whether an automated debugging technique has produced the *correct* fault locations, the *correct* bug diagnosis, or the *correct* patch for an error? We also examine if participants believe that the diagnosis and repair of these bugs can be fully automated and the reasons for their beliefs.

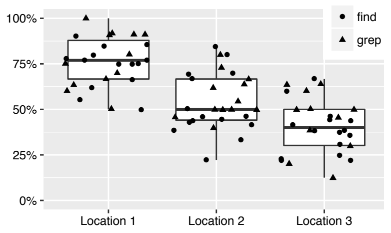


Figure 10: Proportional agreement on Top-3 most suspicious fault locations showing box plots with jitter overlay (each shape is one of 27 errors).

[4]Examples of changing the loop or function flow are adding a return, exit, continue, or goto statement.

[5]Note that one patch can modify several statements!

Consensus on Fault Locations. *For most errors (above the median), more than 75% of participants independently localize the same location as pertinent to explain the error (Location 1).* For each error, we count the proportion of participants independently reporting a fault location. Sorted by proportion, we get the top-most, 2nd-most, and 3rd-most suspicious locations (Location 1–3). For the middle 50% of errors, between half and two third of participants independently report *the same* 2nd-most suspicious location while between one third and half of participants still independently report *the same* 3rd-most suspicious location. However, note that a participant might mention less or more than three locations. The consensus of professional software developers on the fault locations suggests that every bug in DbgBench is *correctly localized* by at least two (2) specific statements. These locations can serve as ground truth for the evaluation of automated fault localization tools.

Consensus on Bug Diagnosis. *10 of 12 participants (85%) give essentially the same diagnosis for an error, on average. In other words, there can be consensus on whether an automated technique has produced a correct bug diagnosis.* These participants are *very confident* (3.7 on the Likert scale) about the correctness of their diagnosis. On the other hand, participants who provide a diagnosis that is different from the consensus are only *slightly confident* (2.4) about the correctness of their diagnosis. The ability to generate a consolidated, concise bug diagnosis that agrees with the majority of diagnoses as provided by professional software engineers shows that understanding and explaining an error is no subjective endeavor. The consolidated bug diagnoses in DbgBench can serve as the ground truth for information that is relevant to practitioners.

Consensus on Bug Fix. *For 18 of 27 bugs (67%), there is at least one other correct fix that conceptually differs from the original developer-provided patch.* In other words, often, there are several ways to patch an error correctly, syntactically and semantically. It might seem obvious that a correct patch can syntactically differ from the patch that is provided by the developer. However, we also found correct patches that conceptually differ from the original patch that was provided by the original developer. For each bug in DbgBench, there are 1.9 conceptually different correct fixes on average. Five (5) bugs (19%) have at least two other conceptually different but correct fixes. For instance, to patch a null pointer reference, one participant might initialize the memory while another might add a null pointer check. To patch an access out-of-bounds, one participant might double the memory that is allocated initially while others might reallocate memory only as needed. For the error in grep.9c45c193 (Fig. 1), some participants remove a negation to change the outcome of a branch while others set a flag to change the behavior of the function which influences the outcome of the branch.

Automation of Bug Diagnosis. *Most professional software developers do not believe that the diagnosis of the errors in DbgBench can be fully automated.* The boxplot in Figure 11 provides more details. For the middle 50% of errors, one to two third of participants believe that bug diagnosis can be automated. However, this varies with bug type. For instance, for 5 of 7 crashes, more than three quarter of participants believe that the crash can be explained intuitively. Functional bugs seem much more involved and intricate such that for the median functional bug only one third of participants believe that it will ever be explained intuitively by a machine.

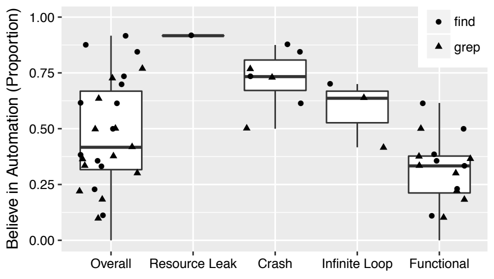


Figure 11: Distribution over all errors (of a certain type) of the proportion of participants who believe that a certain error may ever be *explained intuitively* by a machine. Each shape in the jitter plot overlay represents one error.

Automation of Program Repair. *For the median bug in DbgBench, only a quarter of participants believes that it can be fixed reliably by a machine* (cf. Figure 12). Again this differs by bug type. While half the participants would still think that 4 of 7 crashes can be fixed reliably by a machine, functional bugs are believed to be most difficult to be fixed automatically. The single resource leak appears to be most easy to fix reliably.

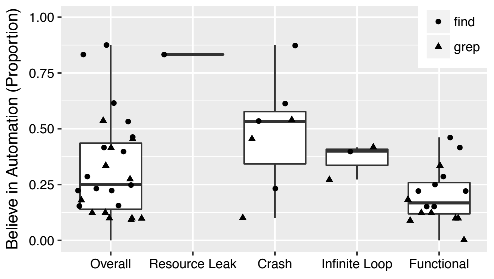


Figure 12: Distribution over all errors (of a certain type) of the proportion of participants who believe that a certain error may ever be *fixed reliably* by a machine. Each shape in the jitter plot overlay represents one error.

Points in Favor of Automation. *Participants believe in automation of bug diagnosis and repair primarily due to the following reasons: (i) sophisticated static or dynamic analysis, (ii) the possibility to check contracts at runtime and (iii) the possibility to cross check with a passing execution.* For instance, for the *resource leak*, where most participants agree that an automated diagnosis and patch is achievable, they reflect on the possibilities of dynamic or static analysis to track the lifetime of resources and to discover the right location in the code to release the resources. For the seven *crashes*, many participants believe in automation via systematic analysis that tracks changes in variable values and explains the crash through these changes. Intuitively, this captures the mechanism employed in dynamic slicing. Besides, participants think of the possibility of having contracts (e.g. a range of expected variable values) in the source code and checking their satisfiability during a buggy execution to explain the error. For *functional bugs*, the most common rationale in favor of automation was due to the possibility to compare a buggy execution with a passing execution.

Points Against Automation. *The majority of participants did not believe in automation due to the lack of a complete specification and due to the difficulty in code comprehension.* For *functional bugs*, most participants think that an automated tool cannot explain such bugs intuitively. This is because such tools are unaware of correct behaviours of the respective program. Similarly, for automated program repair, participants think that it is impossible for a tool to change or add any functionality to a buggy program. Moreover, even in the presence of a complete specification, participants do not believe in automated repair due to the challenges involved in code comprehension (e.g. the meaning of a variable or statement in the code). Finally, the difficulty to analyze side-effects of a fix is also mentioned as a hindrance for automated bug repair.

4 A BENCHMARK FOR DEBUGGING TOOLS

As our study participants agree on so many points, one can actually treat their joint diagnosis and other bug features as *ground truth:* For each bug, the joint diagnosis and fix is what a debugging tool should aim to support and produce. To support *realistic evaluation* of automated debugging and repair approaches, we have compiled a benchmark named DbgBench, which encompasses the totality of our data. Using the example in Figure 1, we illustrate how DbgBench (and thus the results of our study) can be used to evaluate fault localization and automated repair tools.

4.1 Evaluating Automated Fault Localization

Statistical fault localization techniques produce a list of statements ranked according to their suspiciousness score. We used *Ochiai* [1] to compute the suspiciousness score for the statements in the motivating example (Figure 13). In order to evaluate the effectiveness of a fault localization (FL) technique, researchers first need to identify the *statements which are actually faulty* and determine the rank of the highest ranked faulty statement (a.k.a. wasted effort [49]).

Ordinal Rank	(1)	(2)	(3)	(4)	(5)	(6)
Line in parser.c	3055	3057	3058	3061	3062	3067
Ochiai Score	0.98	0.98	0.98	0.98	0.98	0.98
Ordinal Rank	(7)	(8)	(9)	(10)	(11)	(12)
Line in parser.c	3094	3100	3103	3107	**3109**	3112
Ochiai Score	0.98	0.98	0.98	0.98	**0.98**	0.98

Figure 13: Top-12 most suspicious statements in file parser.c. There are 26 statements with the same suspiciousness (0.98), including parser.c:3109 mentioned by our participants.

Using DbgBench, researchers can use the *actual* fault locations that practitioners point out. For instance, in Figure 1-b, the highest ranked *faulty* statement in DbgBench is parser.c:3109. As it turns out, this statement is also within the set of most suspicious statements with an Ochiai-score of 0.98. Thus, DbgBench provides a useful artifact to validate the effectiveness of FL techniques.

Without DbgBench, the "faulty" statements are typically identified as those statements which were changed in the original patch to fix error [7, 13, 49]. However, this assumption may not always hold [42]. Figure 14 shows the original patch for our example. *No statement* was changed in the buggy version. The original patch merely introduced new statements. In fact, only 200 of 290 patches (69%) submitted by our participants modify at least one statement that is referenced in the consolidated bug diagnosis.

```
static boolean get_relative_timestamp (const char *str, ...)
       if (get_comp_type(&str, ...))
...
static boolean parse_time (...)
       const char *timearg = argv[*arg_ptr];
    +  const char *orig_timearg = timearg;
...
       if(get_comp_type(&timearg, &comp))
...      ...
    +  timearg = orig_timearg;
       if (!get_relative_timestamp(timearg, ...))
         return false;
```

Figure 14: Original developer-patch for bug in Figure 1.

4.2 Evaluating Automated Program Repair

Automated Program Repair (APR) techniques automatically generate a patch for a buggy program such that all test cases pass. We ran Relifix [40] to generate the following patch for our motivating example. The generated patch directly uses the original value assigned to `timearg` before calling `get_comp_type` the second time. This is in contrast to the developer-provided patch in Figure 14 which copies `timearg` and restores it after the first call to `get_comp_type`. Is the auto-generated patch correct?

```
static boolean parse_time (...)
    -  if (!get_relative_timestamp(timearg, ...))
    +  if (!get_relative_timestamp(argv[*arg_ptr], ...))
         return false;
```

Using DbgBench, we can evaluate the *correctness* of the auto-generated patch. First, for many participant-provided patches we provide new test cases that fail for an incorrect patch even if the original test cases all pass. For instance, we constructed a new test case for the incorrect-patch-example in Figure 1-c, where the actual output of `./find -mtime +n` is compared to the expected output. Executing it on Relifix's auto-generated patch above, we would see it passes. We can say that the auto-generated patch does not make the same mistake. While we can still not fully ascertain the correctness of the patch, we can at least be confident the auto-repair tool does not make the same mistakes as our participants.

However, to establish patch *correctness* with much more certainty and to understand whether practitioners would actually *accept* an auto-generated patch, we suggest to conduct a user study. Within such a user study, DbgBench can significantly reduce the time and effort involved in the manual code review since the available bug diagnosis, simplified and extended regression test cases, the bug report, the bug diagnosis, fault locations, and developer-provided patches are easily available. For instance, while Relifix's auto-generated patch conceptually differs from the original in Figure 14, it is easy to determine from the provided material that the auto-generated patch is in fact correct.

DbgBench includes time taken by professional software engineers to fix real-world software bugs. We can use these timing information to evaluate the *usefulness* of an automated program repair tool. To this end, we can design an experiment involving several software professionals and measure the reduction in debugging time while using an automated program repair tool.

4.3 DbgBench Artifact

DbgBench was given the *highest badge* by the ESEC/FSE Artifact Evaluation Committee. DbgBench is the first human-generated benchmark for the qualitative evaluation of automated fault localization, bug diagnosis, and repair techniques.

Objectives. The objectives of the DbgBench artifact are two-fold:

(1) To facilitate the sound replication of our study for other researchers, participants, subjects, and languages, we publish our battle-tested, formal experiment procedure, and effective strategies to mitigate the common pitfalls of such user studies. To the same effect, we publish tutorial material (videos and slides), virtual infrastructure (Docker), and example questionnaires which elicit the studied debugging artefacts.
(2) To facilitate the effective evaluation of automated fault localization, diagnosis, and repair techniques w.r.t. the judgement of human experts, we publish all collected data. This includes the fault locations, bug diagnoses, and patches that were provided by the practitioners. For each error, it also includes the error-introducing commit, the original and a simplified bug report, a test case failing because of the error, and the developer-provided patch. Moreover, this artefact contains our reconciled bug diagnoses, our classification of the patches as correct and incorrect together with the general fixing strategies, and which rationale we have to classify a patch as incorrect.

Provided data. Specifically, we make the following data available:

- The *benchmark summary* containing the complete list of errors, their average debugging time, difficulty, and patch correctness, human-generated explanations of the runtime actions leading to the error, and examples of correct and incorrect fixes, sorted according to average debugging time.
- The *complete raw data* containing the responses to the questions in Figure 3 and Figure 4 for each debugging session:
  - the error ID to identify error,
  - the participant ID to identify participant,
  - the timestamp to follow participants accross errors,
  - fault Locations, bug diagnosis, and patches,
  - the confidence in the correctness of their diagnosis or patch,
  - the difficulty to diagnose or patch the error,
  - the time taken to diagnose or patch the error,
  - what would have helped reducing diagnosis or patch time,
  - the steps taken to arrive at the diagnosis or patch,
  - the tools used to arrive at the diagnosis or patch,
  - the problems if they could not diagnose or patch the error,
  - whether he/she believes that generating the diagnosis or patch for this error would ever be automated,
  - why he/she believes so,
  - the code familiarity as it increases over time,
  - the diagnosis techniques used, and
  - how he/she ensured the correctness of the submitted patch,
- The *complete cleaned data* containing for each error:
  - the regression-introducing commit,
  - the simplified and original bug reports,
  - the important fault locations,
  - the reconciled bug diagnosis,
  - the original, developer-provided patch,

  - the patches submitted by participants,
  - our classification of patches as correct or incorrect,
  - our rationale of classifying a patch as incorrect, and
  - test cases that expose an incorrect patch.
- An *example questionnaire* to elicit the raw data above.
- The *Docker virtual infrastructure* and instructions how to use.
- The *tutorial material*, incl. slides, videos, and readme files.

4.4 Disclaimer

DbgBench is a first milestone towards the *realistic evaluation* of tools in software engineering that is grounded in practice. DbgBench can thus be used as necessary reality check for in-depth studies. When conducting user studies, DbgBench can significantly reduce the time and cost that is inherent in large user studies involving professional software engineers. In the absence of user studies, DbgBench allows in-depth evaluations while minimizing the number of potentially unrealistic assumptions. For example, we found a high consensus among participants on the location of the faults. They would often point out several contiguous code regions rather than a single statement. We also found no overlap of fault locations with fix locations. Existing error-benchmarks might assume that an artificially injected fault (i.e., a single mutated statement) or fix locations (i.e., those statements changed in the original patch) are representative of realistic fault locations. DbgBench dispenses with such assumptions and directly provides those fault locations that practitioners would point out with high consensus. DbgBench allows the realistic evaluation of automated debugging techniques that is grounded in practice.

However, we would strongly suggest to also utilize other benchmarks, such as CoREBench [7] or Defects4J [13], for the *empirical evaluation*. Only an empirical evaluation, the use of a sufficiently large representative set of subjects, allows to make empirical claims about the efficacy of an automated debugging technique. Going forward, we hope that more researchers will produce similar realistic benchmarks which take the practitioner into account. To this end, we also publish our battle-tested, formal experiment procedure, effective strategies to mitigate common pitfalls, and all our material. We believe that no single research team can realistically produce a benchmark that is both representative and reflects the realities of debugging in practice. Hence, we propose that similar benchmarks be constructed alongside with *user studies*. Software engineering research, including debugging research, must serve the needs of users and developers. User studies are essential to properly evaluate techniques that are supposed to automate tasks that are otherwise manual and executed by a software professional. Constructing new benchmarks as artifacts during user studies would allow to build an empirical body of knowledge and at the same time minimize the cost and effort involved in user studies.

5 LIMITATIONS

Generalizability of Findings. For the results of our experiment we do not claim generalizability of the findings. We decided on two subjects to limit the time a participant spends in our study to a maximum of three working days and to help participants to get accustomed to at most two code bases. We chose 27 reproducible real errors in single-threaded open-source C projects where bug reports and test cases are available. Our findings may not apply to (irreproducible) faults in very large, distributed, multi-threaded or interactive programs, to short-lived errors that do not reach the code repository, to errors in programs written in other languages, or to errors in programs developed within a software company. We see DbgBench as an intermediate goal for the community rather than the final benchmark.

Hence, we encourage fellow researchers to extend and conduct similar experiments in order to build an empirical body of knowledge and to establish the means of evaluating automated debugging techniques more faithfully, without having to resort to unrealistic assumptions [32] during the evaluation of an automated technique. To facilitate replication, the questionnaires, the virtual infrastructure, and the tutorial material are made available [43]. While we do not claim generality, we do claim that DbgBench is the first dataset for the evaluation of automated diagnosis and repair techniques with respect to the judgment of twelve expert developers. In the future, DbgBench may serve as subject for *in-depth experiments*.

In empirical research, in-depth experiments may mistakenly be taken to provide little insight for the academic community. However, there is evidence to the contrary. Beveridge observed that "more discoveries have arisen from intense observation than from statistics applied to large groups" [21]. This does not mean that research focusing on large samples is not important. On the contrary, both types of research are essential [16].

Cognitive Bias. Results may suffer from cognitive bias since participants fill a questionnaire for each errors, such that all responses are *self-reported* [2, 9, 20]. However, Perry et al. [33] found that self-reports produced by the developers, in their case, often corresponded sufficiently with observations independently recorded by a researcher. As standard mitigation of cognitive bias,

(1) we *avoid leading questions* and *control for learning effects*,
(2) we *reinforce confidentiality* for more truthful responses,
(3) we *use triangulation* [37] by checking the consistency of replies to separate questions studying the same subject,
(4) we mostly *utilize open-ended questions* that provide enough space for participants to expand on their replies
(5) and otherwise *utilize standard measures* of qualitative attributes, such as the Likert scale [29].

While our experiment was fairly long-running, we also suggest to replicate our study at a different point in time with different participants to check whether the responses are *consistent*.

Observer-Expectancy Bias. To control for expectancy bias where participants might behave differently during observation, we conducted the study remotely in a virtual environment with minimal intrusion. Participants were encouraged to use their own tools. We also stressed that there was no "right and wrong behavior".

Imposed Time Bound. We suggested the participants to complete an error in 45 minutes so as to remain within a 20 hours time frame. Some errors would take much more time. So, given more time, the participants might form a better understanding of the runtime actions leading to the error and produce a larger percentage of correct patches. However, participants told us that they felt comfortable to diagnose and patch each error within the stipulated time-bound. They tended to stretch the bounds whenever necessary, taking time from errors that were quickly diagnosed and patched.

6 RELATED WORK

In 1997, Liebermann introduced the CACM special section on THE DEBUGGING SCANDAL AND WHAT TO DO ABOUT IT with the words "Debugging is still as it was 30 years ago, largely a matter of trial and error" [28]. Recently, Beller et al. [4] conducted a substantial mixed methods study involving 600+ developers and found that developers do not use the interactive debugger as often as expected since trace-based debugging (i.e., printf debugging) remains a viable choice. Furthermore, both the knowledge and the use of advanced debugging features are low.

In 2015, Perscheid et al. [34] set out to determine whether the state-of-the-practice had since improved and could only answer in the negative: Debugging remains what it was 50 years ago, largely manual, time-consuming, and a matter of trial-and-error rather than automated, efficient, and systematic. We believe that, at least in part, the debugging scandal is brought about by the absence of user studies and the *assumptions* that researchers had to make when evaluating the output of a machine without involving the human. Only recently, several studies have uncovered that many of these assumptions are not based upon empirical foundations [7, 32, 39, 42, 53]. In this work, we attempt to remedy this very problem.

While several researchers investigated *debugging strategies* that developers generally employ in practice, it remains unclear whether developers who independently debug the same error essentially localize the same faulty statements, provide the same explanation (i.e., diagnosis) and generate the same patch. Perscheid et al. [34] visited four companies in Germany and conducted think-aloud experiments with a total of eight developers during their normal work. While none was formally trained in debugging, all participants used a simplified, implicit form of *Scientific Debugging* [51]: They mentally formulated hypotheses and performed simple experiments to verify them. Katz and Anderson [14] classify bug diagnosis strategies broadly into *forward reasoning* where the programmer forms an understanding of what the code should do compared to what it actually does, and *backward reasoning* where the programmer reasons backwards starting from the location where the bug is observed (e.g., [47]). Romero et al. [36] explore the impact of graphical literacy on the choice of strategy. Lawrance et al. [25] model debugging as a predator following scent to find prey. The authors argue that a theory of navigation holds more practical value for tool builders than theories that rely on mental constructs. Gilmore et al. [11] studied different models of debugging and their assumptions. The authors argue that the success of experts at debugging is not attributed to better debugging skills, but to better comprehension. Our evaluation of the impact of "Code Comprehension" in this paper further validates these findings.

Several colleagues have investigated debugging as a human activity. Ko et al. [19] observed how 10 developers understand and debug unfamiliar code. The authors found that developers interleaved three activities while debugging, namely, code search, dependency analysis and relevant information gathering. Layman et al. [26] investigated how developers use information and tools to debug, by interviewing 15 professional software engineers at Microsoft. The authors found that the interaction of hypothesis instrumentation and software environment is a source of difficulty when debugging. Parnin and Orso [31] also conducted an empirical study to investigate how developers use and benefit from automated debugging tools. The authors found that several assumptions made by automated debugging techniques do not hold in practice. While these papers provide insights on debugging as a human activity, none of these studies provides the data and methods that would allow researchers to evaluate debugging tools against fault locations, bug diagnosis, and patches provided by actual software engineering professionals.

Many researchers have proposed numerous tools to support developers when debugging, but only a few have evaluated these tools with user studies, using real bugs and professional developers. For instance, between 1981 and 2010, Parnin and Orso [31] identified only a handful of articles [10, 18, 22, 38, 46, 48] that presented the results of a user study: Unlike this paper, none of these studies involved actual practitioners and real errors. In our own literature survey,[6] studying whether the state-of-the-research has since improved, we could identify only *three (3) papers* that conduct user studies with actual practitioners and real errors in the last five years. Two articles employed user studies to evaluate the acceptability of the auto-generated patches [8, 15] while one had practitioners to evaluate the effectiveness of an auto-generated bug diagnosis [41].[7] We believe that this general absence of user studies is symptomatic of the difficulty, expense, and time spent conducting user studies. This is the problem we address with DBGBENCH.

7 CONCLUSION

Given how much time practitioners spend on debugging, it still is a scandal how little we know about debugging. Yet, knowing how humans *do* debug is an important prerequisite for suggesting how they *could* debug. With our study, we hope to have shed some light into this under-researched area; with our DBGBENCH benchmark, we hope to contribute some ground truth to guide the development and evaluation of future debugging tools towards the needs and strategies of professional developers. The DBGBENCH benchmark as well as all other study data is available at our project site

https://dbgbench.github.io

ACKNOWLEDGMENTS

We thank the anonymous reviewers in the PC and AEC for their helpful and constructive comments. Next, we thank Shin Hwei Tan for her help with RELIFIX, as well as Sebastian Proksch for his helpful feedback on experiment design. Finally, we greatly appreciate the generous financial support from Saarland University. This work was partially supported by SUTD research grant number SRIS17123.

[6]We surveyed the following conference proceedings: ACM International Conference on Software Engineering (ICSE'11–16), ACM SIGSOFT Foundations of Software Engineering (FSE'11–16), ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'11–16), IEEE/ACM International Conference on Automated Software Engineering (ASE'11–16), and International Conference on Software Testing, Verification and Validation (ICST'11–16) and identified 82 papers on automated debugging or software repair only 11 of which conducted user studies. From these only three (3) involved software engineering professionals *and* real errors.

[7]Tao et al. [41] measured debugging time and the percentage of correct repairs provided by the participants, after they had received an auto-generated patch as bug diagnosis

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. 39–46.

[2] Elizabeth J. Austin, Ian J. Deary, Gavin J. Gibson, Murray J. McGregor, and J.Barry Dent. 1998. Individual response spread in self-report scales: personality correlations and consequences. *Personality and Individual Differences* 24, 3 (1998), 421 – 438.

[3] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 306–317.

[4] Moritz Beller, Niels Spruit, and Andy Zaidman. 2017. How developers debug. *PeerJ Preprints* 5 (2017), e2743v1.

[5] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. 2013. Partition-based Regression Verification. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 302–311.

[6] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. 2013. Regression Tests to Expose Change Interaction Errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. 334–344.

[7] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying Complexity of Regression Errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. 105–115.

[8] Wensheng Dou, Shing-Chi Cheung, and Jun Wei. 2014. Is Spreadsheet Ambiguity Harmful? Detecting and Repairing Spreadsheet Smells Due to Ambiguous Computation. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 848–858.

[9] Xitao Fan, Brent C. Miller, Kyung-Eun Park, Bryan W. Winward, Mathew Christensen, Harold D. Grotevant, and Robert H. Tai. 2006. An Exploratory Study about Inaccuracy and Invalidity in Adolescent Self-Report Surveys. *Field Methods* 18, 3 (2006), 223–244.

[10] Margaret Ann Francel and Spencer Rugaber. 2001. The value of slicing while debugging. *Science of Computer Programming* 40, 2-3 (2001), 151–169.

[11] David J Gilmore. 1991. Models of debugging. *Acta psychologica* 78, 1 (1991), 151–172.

[12] Jerald Greenberg and Robert Folger. 1988. *Experimenter Bias*. Springer New York, 121–138.

[13] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. 437–440.

[14] Irvin R. Katz and John R. Anderson. 1987. Debugging: An Analysis of Bug-location Strategies. *International Journal of HumanâĂŞComputer Interaction* 3, 4 (Dec. 1987), 351–399.

[15] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. 802–811.

[16] Barbara A. Kitchenham and Shari L. Pfleeger. 2008. *Personal Opinion Surveys*. Springer London, London, 63–92.

[17] Andrew J. Ko, Thomas D. Latoza, and Margaret M. Burnett. 2015. A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants. *Empirical Software Engineering* 20, 1 (Feb. 2015), 110–141.

[18] Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. 151–158.

[19] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006).

[20] Justin Kruger and David Dunning. 1999. Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments. *Journal of Personality and Social Psychology* 77, 6 (1999), 1121–1134.

[21] Adam Kuper and Jessica Kuper. 1985. *The Social Science Encyclopedia*. Routledge.

[22] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. 2002. Experimental evaluation of program slicing for fault localization. *Empirical Software Engineering* 7, 1 (2002), 49–76.

[23] Sanford Labovitz. 1967. Some observations on measurement and statistics. *Social Forces* 46, 2 (1967), 151–160.

[24] Thomas D. LaToza and Brad A. Myers. 2011. Designing Useful Tools for Developers. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '11)*. 45–50.

[25] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. 2013. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering* 39, 2 (Feb 2013), 197–215.

[26] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert DeLine, and Gina Venolia. 2013. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, 383–392.

[27] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72.

[28] Henry Lieberman. 1997. The Debugging Scandal and What to Do About It (Introduction to the Special Section). *Commun. ACM* 40, 4 (1997), 26–29.

[29] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[30] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. 691–701.

[31] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA)*. 199–209.

[32] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating & improving fault localization techniques. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*.

[33] Dewayne E. Perry, Nancy Staudenmayer, and Lawrence G. Votta. 1994. People, Organizations, and Process Improvement. *IEEE Software* 11, 4 (July 1994), 36–45.

[34] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2016. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* (2016), 1–28.

[35] Research Ethics Policy and Advisory Committee, University of Toronto. 2017. Guidelines for Compensation and Reimbursement of Research Participants. (2017).

[36] Pablo Romero, Benedict du Boulay, Richard Cox, Rudi Lutz, and Sallyann Bryant. 2007. Debugging strategies and tactics in a multi-representation software environment. *International Journal of Human-Computer Studies* 65, 12 (2007), 992 – 1009.

[37] Paulette Rothbauer. 2008. *Triangulation.* The SAGE Encyclopedia of Qualitative Research Methods, 892–894.

[38] Jonathan Sillito, Kris De Voider, Brian Fisher, and Gail Murphy. 2005. Managing software change tasks: An exploratory study. In *Empirical Software Engineering, 2005. 2005 International Symposium on*. IEEE, 10–pp.

[39] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-based Fault Locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. 314–324.

[40] Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated Repair of Software Regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. 471–482.

[41] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches As Debugging Aids: A Human Study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 64–74.

[42] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 1–11.

[43] Website. 2017. DBGBench: From Practitioners for Researchers. https://dbgbench.github.io. (2017). Accessed: 2017-06-30.

[44] Website. 2017. Docker Infrastructure – Homepage. http://docker.com/. (2017). Accessed: 2017-06-30.

[45] Website. 2017. Upwork - Hire Freelancers and Post Programming Jobs. https://www.upwork.com/. (2017). Accessed: 2017-06-30.

[46] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. 439–449.

[47] Mark Weiser. 1982. Programmers Use Slices when Debugging. *Commun. ACM* 25, 7 (July 1982), 446–452.

[48] Mark Weiser and Jim Lyle. 1986. Experiments on slicing-based debugging aids. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. Ablex Publishing Corp., 187–197.

[49] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (Aug. 2016), 707–740.

[50] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Lamelas Marcote, T. Durieux, D. Le Berre, and M. Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* PP, 99 (2016), 1–1.

[51] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[52] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[53] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*. 913–923.