How Developers Diagnose and Repair Software Bugs (and what we can do about it)

Marcel Böhme* Ezekiel O. Soremekun† Sudipta Chattopadhyay† Emamurho J. Ugherughe† Andreas Zeller†
*National University of Singapore, Singapore, marcel.boehme@acm.org
†Saarland University, Germany, {soremekun,ugherughe,sudiptac,zeller}@cs.uni-saarland.de

***Abstract*—How do practitioners debug computer programs? In a retrospective study with 180 respondents and an observational study with 12 practitioners, we collect and discuss data on how developers spend their time on diagnosis and fixing bugs, with key findings on tools and strategies used, as well as highlighting the need for automated assistance. To facilitate and guide future research, we provide a highly usable debugging benchmark providing fault locations, patches and explanations for common bugs as provided by the practitioners.**

I. Introduction

In the software engineering community, the past decade has seen a surge in automated debugging techniques designed to assist programmers locating and fixing faults in software. A recent survey [1] on automated fault localization cites no less than 427 papers related in some way or another to the problem of determining possible fault locations for a given failure. Yet, we mostly ignore how such tools would address real-world debugging needs. 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. This is unfortunate, as it is feedback from practice that should shape and determine future research in our field.

In this paper, we address this issue by providing *data on how practitioners debug.* Our DBGBENCH benchmark, presented in this paper, allows to evaluate novel automated debugging techniques by providing fault locations, error explanations, fix explanations, and human-generated patches for a set of 27 real world erros in the find and grep programs. In contrast to past bugs and fixes obtained from software archives (e.g., [2]), which typically represent only one problem solution and lack diagnosis and details, our benchmark represents the large variance of how developers debug programs, incorporating different strategies, fixes, and error explanations—including correct and incorrect ones. In other words, DBGBENCH captures the reality of today's debugging—and this is what debugging tools should address.

To obtain DBGBENCH, we have ran two major studies, described in this paper: In an initial *retrospective study,* 180 respondents (including the study participants) provided insights in their general debugging process, giving additional insights about the state of the practice in debugging, and how the present and future state of the art might help in addressing these problems. This shaped our *observational study* with 12 practitioners, having them spend 29 working days on debugging 27 real errors in open-source C programs. Each participant would be given an executable, failing test case and a simplified bug report. As an example, consider Figure 1, showing a bug report for the `grep` program. Given this bug report, we would ask for the practitioners to provide a fix (i.e., debug the program), measuring, among others, the time spent on bug reproduction, diagnosis, and fixing; and asking them, among others, for their familiarity with the code, the difficulty of the task, and the strategies they used for finding the fault. All this is contained in DBGBENCH.

```
Hang in grep -F for empty string search
Searching with grep -F for an empty string in a
multibyte locals would freeze grep.
For example,
$ export LC_ALL=en_US.UTF-8
$ echo "abcd" | ./grep -F ""
(runs forever)
```

Fig. 1. `grep.5fa8c7c9` bug report

In this paper, we also use the DBGBENCH data to *investigate a number of common assumptions about debugging techniques,* including *tools and strategies used* (the majority of developers always or often relies on traces and interactive debuggers, but never uses slicing, algorithmic debugging, or statistical debugging), the *location and span of faults* (explanations typically span multiple functions, whereas patches tend to be local to one function), *characteristics of patches* (every patch applies about two changes to modify either data or control flow—in contrast to mutation analysis or automated repair, where patches are much less complex), whether developers *fix programs in the best way possible* (a third of patches produced treat symptoms rather than causes), and how practitioners would *design debugging tools* (56% want tools that describe the actions or conditions leading to the error, or the deviation from an expected execution).

The remainder of this paper is organized as follows. After discussing the background (Section II), Section III details our retrospective study and its results. Section IV discusses our observational study, detailing how practitioners spend their time on diagnosing and fixing bugs, and again discussing the results. In Section V, we present the debugging benchmark with all data resulting from the study. After discussing limitations and threats to validity (Section VI), Section VII closes with conclusion and consequences.

II. Background

Debugging is one of the most difficult and time consuming activities in the software development process [1]. In the past, several works have studied the effectiveness of automated debugging assistants, such as automated fault localization, in practice. For instance, Parnin and Orso [3] found that a ranked list of suspicious statements does not help towards a faster and better bug diagnosis. Participants spent up to 23 minutes debugging one of two errors with and without the help of a statistical fault localization tool. To investigate how statistical fault localization might be improved, Kochhar et al. [4] asked practitioners about their expectations of automated fault localization. The study explored several crucial parameters, such as trustworthiness, scalability and efficiency, in order for a practitioner to adopt a statistical fault localization tool. These works provide valuable insights on potential research directions in automated debugging. However, we find two major limitations of the current literature that study debugging activities in practice:

- The authors focus on a specific debugging technique and not the debugging activity in general, and
- There is a general lack of studies that focus on developers debugging *real programming errors.*

This motivates us to initiate our work that studies the activity of debugging real software errors. By designing such a study, we discover that debugging real software error requires better fault localization and repair tools. Besides, we confirm that practitioners rarely use any automated debugging tools (*e.g.* slicing or statistical debugging). This highlights potential actions that need to be taken, in order to bring the research of automated debugging into practice.

There are several works that investigate and model debugging strategies in practice. Perscheid et al. [5] study available literature, the tool support, and debugging strategies used in practice. The authors visited four companies in Germany and conducted think-aloud experiments with eight developers at during their normal work. Romero et al. [6] explore the impact of verbal ability and the level of graphical literacy on the choice of debugging strategy and debugging performance. Finally, Lawrance et al. [7] model debugging using information foraging theory. In contrast, our focus is not on assessing the current state-of-practice. Instead, we investigate whether common assumptions in automated debugging research hold in practice. We also investigate manual debugging strategies with the aim of identifying how automated debugging assistants may support the debugging task.

A primary motivation of our study is to provide the research community a set of real errors with human-generated root diagnoses and patches. Existing benchmarks, which collect real software bugs (*e.g.* [2], [8], [9], [10]), do not provide information on human-generated root causes, patches and time spent in bug diagnosis and repair. In order to bridge this gap, we provide such information in the benchmark created as an artifact of this paper. We hope that such a benchmark will open up several research directions in the future.

III. Retrospective Study

We start with our retrospective study, which we conducted to obtain a general impression on today's practice of debugging.

A. Study Design

Study Objective. The main objective of the retrospective study is to explore the task of debugging in software engineering practice and elicit challenges and opportunities for researchers to automate the process. Practitioners are asked about several aspects of their day-to-day debugging activities. We focus our exploration on the following research questions.

- *Time and Familiarity*. How much time do practitioners spend debugging. How familiar are they with the debugged code?
- *Techniques Used*. Which tools and techniques do practitioners use? Is debugging perceived to be systematic or trial-and-error?
- *Techniques Needed*. Which techniques should be developed? Which tool output would be considered most helpful?

Terms. We distinguish three distinct tasks of debugging. *Bug reproduction* is the task of reproducing the bug locally and finding a test case that confirms the unexpected behavior. *Bug diagnosis* is the task of understanding and explaining the runtime actions that lead to the unexpected behavior. Finally, *bug fixing* is the task of removing the error. The *symptom* of an error is the deviation of the actual from the expected program output or behavior for a given test input (e.g., a program crash).

Survey. Respondents filled an online questionnaire which begins with informing respondents about the goals of our study and some basic terminology. We requested general demographic information about occupation, experience, and skill. After the technical questions investigating the study objectives, we allow participants to register for the observational study. Otherwise, both studies are completely anonymous.

Measures. In order to quantify attitudes to a topic, such as code familiarity, we use the common *5-point Likert scale* [11]. This 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. When asking how often participants use certain *techniques*, we offer the following choices:

[ ] Trace-based Debugging (using printing; e.g., println, log4c)
[ ] Interactive/Online Debugging (using breakpoints; e.g., gdb, jdb)
[ ] Post-Mortem/Offline Debugging (using core dumps, stack traces)
[ ] Regression Debugging to find faulty changes (e.g., git bisect)
[ ] Statistical/Spectrum-based Debugging to find faulty statements (e.g., Tarantula)
[ ] Program Slicing (e.g., Frama-C, CodeSurfer)
[ ] Time Travel or Reversible Debugging (e.g., UndoDB)
[ ] Algorithmic or Declarative Debugging (e.g., JavaDD)

Demographics. We advertised both studies on several freelancer platforms and social as well as professional networks, including Upwork, Guru, Freelancer and Github. The *majority of respondents are professional software developers with seven (7) years or more experience* in software development rating their level of skill as *advanced or expert*. One in four respondents is a student and one in six is a researcher. A quarter has three to six years of experience and the remaining 22% of respondents have two years or less of experience. One in three respondents rate their level of skill as intermediate.

B. Results

The study ran over *14 months* and gathered *180 entries*.

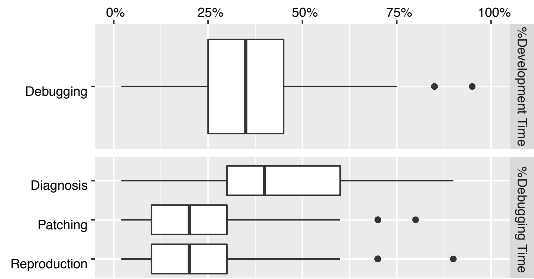

Fig. 2. Distribution of development and debugging time.

Debugging Time & Code Familiarity. Respondents spend about *one third* of their development time with debugging (see Figure 2). During debugging, they spend *half their time with bug diagnosis*. Respondents spend about as much time trying to reproduce an error from a bug report as they spend patching the error. Most respondents did not write the software they are debugging. We asked to rate code familiarity and how often they debug other people's code on a 5-point Likert scale. About *two in three respondents are moderately or less familiar* with the code she debugs. About *every second respondent often or always* debugs code that is not her own.

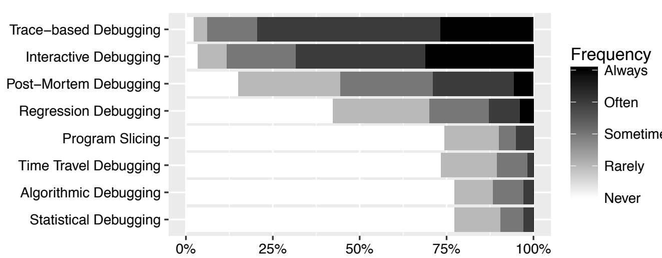

Fig. 3. Debugging Techniques and their Frequency

Techniques Used. In practice, bug diagnosis is still a vastly manual effort. As shown in Figure 3, most respondents *always or often* use techniques such as trace-based debugging (e.g., println) and interactive debugging (e.g., gdb). They *sometimes* use post-mortem debugging (e.g., inspecting coredump and stack traces). They *rarely* use regression debugging (e.g., git bisect). The majority of respondents *never* used any of the remaining choices. Respondents mentioned memory, coverage, and performance profiler and analysis tools, such as valgrind, gcov, and gprof as additional tools which they use, but which are not listed. One in three respondents admits to trial-and-error versus a more systematic debugging approach.

Techniques Needed. In practice, most respondents would like to design and use a tool that does what has already been achieved in automated debugging research. For instance, *two in five (40%) respondents would like a tool that points out suspicious statements* or functions in the source code while almost nobody has ever used statistical debugging. However, *two in five (40%) respondents would design a tool that outputs a sophisticated bug diagnosis* rather than only fault locations. These respondents asked for a high-level or an approximate explanation of the pertinent sequence of events leading to the error – perhaps as cause-effect chains or even as an English narrative. *One in eight (13%) respondents would output an auto-generated patch* in lieu of a diagnosis. The same percentage would output the most general conditions under which the error occurs (e.g., input range or OS dependence). Moreover, respondents prefer a tool that generates a short, high-level bug diagnosis versus a long, detailed one. Minor, simple bugs should have only small explanations. Most respondents would output some information about the program state, such as variable values and possibly how and where the observed state deviates from the expected state.

Specifically, we asked which output an automated diagnosis assistant would provide if the respondent designed the tool. We used *open card sort* [12] to establish the categories and quantify the prevalence. If we did not find related work that addresses a practitioners' need, the concern is shown in **bold.**

In terms of general *program comprehension*, developers are interested in tools that

- (7%) visualize the value history of a variable [13],
- (3%) visualize data structures and allow to persist, restore, and compare their states [13],
- (3%) help at program understanding, generate documentation [14],
- (1%) uncover "meaning" of variables and value range [15], [16],
- (1%) point to code fragments processing certain input bytes [17],

In terms of *automated bug diagnosis*, developers are often interested in tools that

- (28%) **generate a diagnosis or explanation why the error occurs**,
- (23%) **report the most general environment or conditions under which the bug can be reproduced**,
- (19%) print the sequence of executed functions for a failing input (can be obtained automatically via any online-debugger),
- (17%) generate a suggestion where and how to patch the bug [18],
- (17%) point to the cause-effect chain leading to the symptom [19],
- (15%) point to suspicious functions or program statements [20],
- (13%) generate a patch to assist in understanding the error [21],
- (7%) **visualize divergence from the *expected* value of a variable**, or
- (1%) **visualize the range of *expected* values for a given variable**.

In terms of *explaining and classifying symptoms*, developers are interested in tools that

- (19%) **highlight the symptoms and side-effects of an error**,
- (8%) **classify the error according to its symptom in a category** (e.g., if nullpointer deref., suggest check or where to init.),
- (1%) **evaluate criticality of the symptoms (e.g., security risk)**,
- (1%) **find program statement that prints the unexpected output**,
- (1%) track allocated resources and where they are allocated/used (as can be obtained via tools like *valgrind* [22]).

Ensuring Patch Correctness. We also asked how respondents make sure that a submitted patch is a correct one. Almost everybody (95%) tries to reproduce the bug on the patched version while more than half generate new test cases and execute the existing regression test suite (57% and 54%, resp.). Only few respondents suggested to rely on a valid bug diagnosis. Several respondents mentioned third-party code review as the best way to ensure patch correctness.

C. Implications

Exigency of Automation. Developers spend ten minutes of every development hour trying to understand[1] the runtime actions leading to an error in a rather unfamiliar program that is often written by somebody else. There is almost no automation in debugging practice. The most frequently used debugging techniques are manual and ad-hoc rather than systematic. At the same time, developers are calling for automated assistants that help with program comprehension and bug diagnosis.

Need and Knowledge. In practice, most practitioners have never used statistical fault localization tools. Yet, two in five (40%) practitioners would like a bug diagnosis tool that can point out suspicious statements of functions. Therefore, we suggest to make automated debugging research more useful to practitioners by considering more carefully how to integrate our research prototypes into the existing development process and environment.

Disconnect and Revision. In research, statistical fault localization is one of the most popular techniques for automated debugging. Then, why do practitioners never use statistical fault localization tools while there is such an evident need? One reason might be flaws in our assumptions. For instance, to evaluate fault localization techniques, faults are artificially injected by changing one statement. Then, a tool is considered effective if it localizes *that* statement with high precision. In the subsequent observational study, we set out to investigate several of these assumptions. For instance, we determine how many locations *practitioners* point out when explaining the pertinent runtime actions *to us*. Another reason might be that most practitioners (56%) would design a bug diagnosis tool that goes beyond simple fault localization. Such a tool would describe the unfortunate chain of events leading up to the error, the deviation from an expected execution, or general conditions under which the error occurs. Many practitioners (13%) would also use an automated patching tool for diagnosis.

Opportunities. Techniques that can derive an approximate English narrative explaining the context and chain of events leading to the error may be very successful in practice. A tool that explains and classifies the symptoms of an error may be as helpful as a tool that can derive the most general environment or conditions under which an error occurs. Moreover, it may be worthwhile to develop debugging tools that can distinguish expected from actual values.

IV. Observational Study

The insights from the retrospective study shaped the design of our observational study. Seeking to understand more about the disconnect between practice and research, we designed and conducted experiments with professional software developers to find out how they debug programs. We used the insights from this observational study constructively and developed the first human-generated benchmark for the evaluation of automated debugging techniques.

[1]Respondents spend 36% of their development time with debugging tasks and 47% of their debugging time with bug diagnosis, on average.

A. Study Design

Study Objectives. The observational study has three main objectives: i) to study more closely how practitioners debug a number of real errors in C programs; ii) to investigate common research assumptions about debugging in practice; and iii) to establish a human-generated benchmark for the evaluation of novel debugging assistants. Participants are given a virtual environment with several buggy versions of the same program. For each version, participants fill an online questionnaire. We focus our investigations on the following research questions.

- *Difficulty, Time, and Familiarity*. How difficult do participants perceive the debugging of certain errors and what makes very difficult errors so difficult? How much time do they spend on bug diagnosis and patching? Does the increasing familiarity with the code affect the likelihood to produce a correct patch?
- *Strategies*. Which steps do participants take to diagnose the bug? What are the ingredients of a developer patch?
- *Single Fault Assumption*. Do participants reference a single statement or one contiguous region when explaining the error?
- *Single Diagnosis Assumption*. Do participants agree on an explanation of the pertinent runtime actions leading to the error, or do they come up with different explanations?
- *Single Patch Assumption*. Do participants agree on a correct patch for an error or do they submit conceptually very different (yet correct) patches?
- *Correctness and Plausibility*. Do participants submit patches that are technically incorrect but plausible (pass the test case)?
- *Fix Location* $\cup$ *Fault Location* $\neq \emptyset$. Do developers fix the same code that they reference in the bug diagnosis?

The benchmark contains the following artifacts:

- *Fault Locations*: We provide the pertinent locations referred to in an effective bug diagnosis.
- *Bug Diagnosis*: We provide a concise explanation of the runtime events leading to the error which references pertinent functions, variables, and data flows.
- *Correct vs. Plausible Patches*: We provide examples of correct and incorrect but plausible patches (the failing test case passes) and an explanation of the changes needed to fix the error.

Infrastructure [23]. To conduct the study remotely and in an unsupervised manner, we developed a virtual environment based on Docker. We prepared *1 readme, 34 slides, and 10 tutorial videos* (~2.5 minutes each) to explain the goals of our study and provide details about subjects and infrastructure. The *virtual environment* is a lightweight Docker image with Ubuntu 14.2 Guest OS containing a folder for each buggy version of either grep or find. A script generates the ID for the participant's responses and scrambles the order of the folders: The first error for one participant might be the last error for another. The image contains most common development and debugging tools, including gdb, vim, and Eclipse. Participants are encouraged to install their own tools and copy the created folders onto their own machine.

Real Errors. We chose *all 27 reproducible errors in find and grep* from CoREBench [2]. The command line tools find and grep are well-known, well-maintained, and widely-deployed open-source C programs. The code bases of find and grep has *17k and 19k lines of code*, respectively. For each error, we provide a failing test case, a simplified bug report, and a large regression test suite.

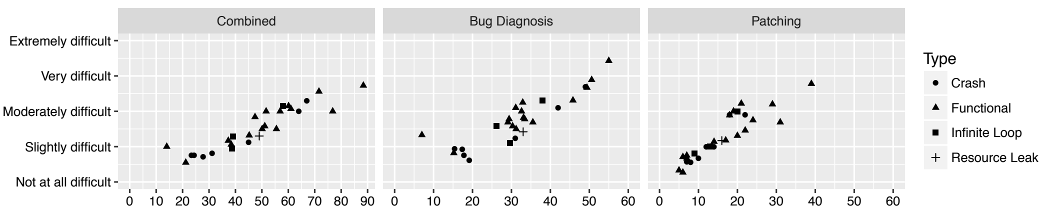

Fig. 4. Average time spent for and the perceived difficulty of explaining and patching each error.

Figure 1 shows a simplified bug report describing inputs, symptoms, and the expected behavior. 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.

Pilot Study (Students). To test our infrastructure and get a first estimate of the time spent in the observational study, we conducted a small supervised version of this study where we invited five (5) student volunteers to our lab. These volunteers were chosen from a larger pool of interested candidates as those with most experience in software development. However, in *seven hours* our student participants submitted only *a sum total* of five patches. The setup and infrastructure was working very well indeed but we felt strongly advised to recruit software engineering professionals for the main study.

Main Study (Professionals). The candidates registered via the questionnaire of the retrospective study. From 180 responses, 130 indicated interest. We selected and invited 89 candidates with sufficient experience in C development. However, only 12 participants actually entered and completed the observational study. These are *one researcher* and *eleven professional software engineers* from six countries (Russia, India, Slovenia, Spain, Canada, and Ukraine). Nine participants have *more than 7 years* experience in developing C programs. All entered C or C++ as their favourite programming language. Upon completion, a participant received 540 USD in compensation for their time and efforts.

B. Results

Overall, *12 participants* spent *29 working days* debugging *27 real errors* in 2 open-source C programs: find and grep.

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. The details are shown in Figure 4. For each error, we asked participants to provide the time spent in understanding the runtime actions leading to the error (bug diagnosis) and in changing the source code so as to remove the error (bug fixing). We also asked to rate the difficulty of both tasks on a 5-point Likert scale. For all errors, a participant spent on average *14 hours 20 minutes* to understand the errors and come up with the diagnosis and *7 hours 11 minutes* to remove the errors and come up with the patches.[2] Developers that work with novel debugging assistants are expected to improve on this time.

[2]In all cases, the time excludes the time spent filling the questionnaire.

Why are some errors very difficult? There are four errors (3 functional, 1 crash) rated as *very difficult* to diagnose which took between 1 and 1.5 hours to debug, on average. 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 is caused by a corrupted heap such that the crash location and that location where heap is corrupted are very distant. 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 patch is actually very complex, involving eighty added and thirty deleted source code lines. Only one participant provided a correct patch.

Code Comprehension. In the retrospective study, we found that many practitioners frequently debug code which they did not write. In the observational study, we investigate the impact of increasing familiarity with the code base as they continue to debug a randomized sequence of errors in the same code base. This allows us to observe trends of the participant as she understands the code with each new error she debugs while controlling for that bias in other analyses. We take the number of submissions for that subject and participant as a measure of her code comprehension.

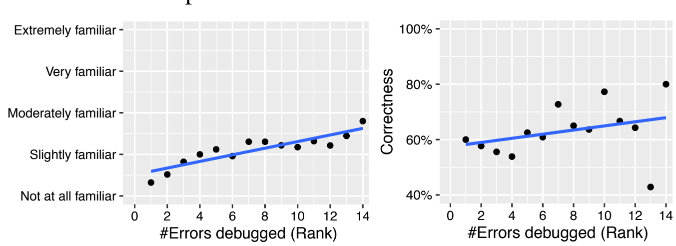

Fig. 5. Patch correctness and code familiarity increase as the developer diagnoses and patches more errors (i.e., as code comprehension increases).

A participant that has a better understanding of the code is more likely to produce a correct patch (Spearman's $\rho = 0.52$). Moreover, participants *who submitted a correct patch spent 25% (5 min) more time* generating the bug diagnosis compared to participants who submitted an incorrect patch. We also asked the participants to rate their code familiarity on a 5-point Likert scale. Unsurprisingly, we found a *very strong correlation between code comprehension and familiarity* (Spearman's $\rho = 0.89$). Both relationships are shown in Figure 5.

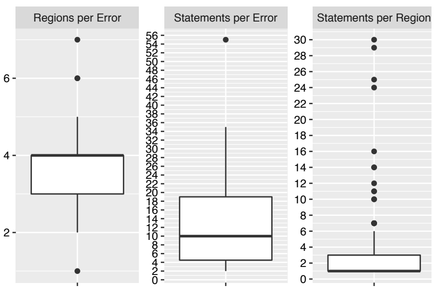

Fig. 6. #Program locations referenced in a bug diagnosis.

Single Fault Assumption. *Most narratives of the pertinent chain of events that lead up to the error reference three to four contiguous regions in the source code*. This contradicts a common assumption in works on automated fault localization and supports a recent finding by Orso and Parnin [3]: It is insufficient to show a developer a suspicious statement for her to understand the error. The set of program statements must be *linked* to the pertinent runtime actions leading to the error. Figure 6 shows more details about the number of pertinent program locations. The middle 50% of bug diagnoses[3] reference three to four contiguous regions in the source code. In most cases, these regions are distributed across different functions and files. *Most regions contain only a single statement*. However, the fourth quartile (i.e, upper 25%) ranges from 3 to 30 statements that constitute a region. Generally, *most bug diagnoses reference 10 statements or less*.

Single Diagnosis Assumption. *85% of participants provide essentially the same diagnosis for an error.*[4] In other words, there are no two participants who are confident about the correctness of their individual diagnosis – which also vastly disagree. For each error, we asked participants to provide the root cause of the error and explain the runtime actions leading to the error while referencing the pertinent locations in the source code. Subsequently, we aggregated these explanations into a single, self-containing, and precise *bug diagnosis*.[5] The ability to extract an agreeable diagnosis shows that the understanding and explanation of an error is no subjective endeavor. The extracted bug diagnoses can serve as the ground truth for information that is perceived relevant to a practitioner.

Patch vs. Fault Location. Only *69% of submitted patches modify statements that are referenced in the bug diagnosis*. However, *unlike an explanation that sometimes stretches over several files, the patch tends to be local to one function*. An assumption of automated debugging research is that the fault and fix location overlap. For instance, an operator fault is fixed by substituting the faulty by the correct operator. However, we often observe the opposite. For instance, the resource leak in grep is explained by pointing out where the resource is opened and used. Finding the location where the resource can be released is another matter. Often new branches, assignments, or function calls are added, for instance, to conditionally print a message, reset a variable, or free/allocate some memory.

Single Patch Assumption. *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 developer-provided patch. 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 one error in grep, 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.

Correctness and Plausibility. *While 282 out of 291 (97%) of the submitted patches pass the test case, only 170 patches (58%) are actually correct*.[6] We determined *patch correctness by code review* and *patch plausibility by executing the provided test case*. A patch is *incorrect* if we can provide an explanation as to *why* it is incorrect. Figure 7.a) shows that for the majority of bugs 69% or less of submitted patches are correct while for the same majority all (100%) submitted patches are plausible. Figure 7.b shows that more than half of the incorrect patches actually introduce new errors and that incorrect patches are incomplete fixes or are treating the symptom in roughly equal parts (20%). A *regression* breaks existing functionality; we could provide a test that fails but passed before. 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. 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. An *incorrect workaround* changes an artifact that is not supposed to be changed, like a third-party library.

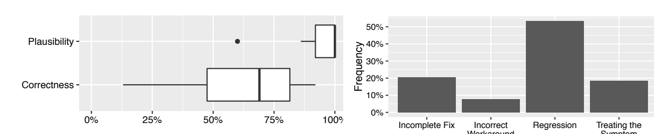

Fig. 7. (a) Average patch correctness and plausibility for an error (left). (b) Reasons for incorrectness (right).

Ensuring Correctness. We asked how participants made sure that the submitted patch is a correct one. According to them, a quarter patches (24%) were checked by generating new test cases while one in ten patches (10%) were checked by executing existing test cases in the regression test suite. Only three in four (72%) patches were checked by executing the failing test case on the patched version. However, some might simply not have mentioned this since we explicitly suggested to execute the failing test case. For one in five patches (19%) participants suggested to rely on intuition and a valid bug diagnosis.

[3]The inter-quartile range (i.e., box) represents the middle 50% of a group.

[4]We note that the disagreeable participants provide a different explanation, about the correctness of which they are only *slightly confident*, on average. In contrast, participants with an explanation that agrees with our diagnosis are *very confident* (3.7) in the correctness of their explanation, on average.

[5]An example of a bug diagnosis can be found in the appendix.

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

C. Bug Diagnosis Strategies

For each error, we asked participants which concrete steps they took to understand the runtime actions leading to the error. We observed the following bug diagnosis strategies.

Classification. We extend the bug diagnosis strategies that have been identified by Romero and colleagues [6], [24]:

- (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 core-dump (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 [25] which might again reflect the potential lack of knowledge about automated techniques that have been available from the research community for over a decade.

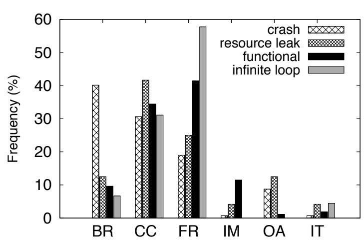

Fig. 8. Frequency of diagnosis strategies for different error types.

Frequency. *Forward reasoning and code comprehension (FR+CC) are the most frequently used* diagnosis strategies. The frequency of bug diagnosis strategies is shown in Figure 8 for the different error types. We also observe that *past experience (IT) is used least frequently*. Therefore, we consider the set of diagnosis strategies to be representative for debugging real and unknown errors. Many participants used input modification (IM) as diagnosis strategy. Therefore, the integration of automated techniques that implement IM (*e.g.* [25]) into mainstream debugger will help improve debugger productivity.

Error Type. We can see in Figure 8 that *most infinite loops (58%) are diagnosed with forward reasoning (FR)*. Intuitively, there is no last executed statement which can be used to reason backwards from. *Two in five crashes (40%) are diagnosed with backward reasoning (BR)*. Intuitively, the crash location is most often a good starting point to understand how the crash came about. *Two in five functional errors (40%) are diagnosed with 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. Input modification is used for 10% of functional errors 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 [25], albeit with manual effort, to narrow down the pertinent sequence of events.

D. Repair Ingredients

Participants submitted a sum total of 291 patches where one third (34%) exclusively affects the control-flow, one third (30%) exclusively affects the data-flow, and the remaining patches (36%) affect both.

Control-Flow. In automated repair research, the patching of control-flow is considered tractable because the search space is binary [26]: Either a set of statements is executed or not. The frequency with which participants patch the control-flow provides some insight about the effectiveness of such an approach. The control-flow is modified by seven in ten patches (69%). Of all the patches that affect the control-flow, a branch condition is changed by 63%. The loop or function flow is modified by every fifth patch that affects the control-flow (19%).[7] A new if-or-else branch is added by two of every five patches affecting the control-flow (43%). In many cases, an existing statement is then moved into the new branch or a new function call is added.

Data-Flow. The data-flow is modified by two of every three patches (64%). Of all patches that change the data-flow, a variable value or function parameter is changed by 30%. The RSRepair [27] and GenProg [28] automated repair systems copy and move existing program statement while Kali [29] effectively deletes existing statements. In our study, participants add, move, or delete a statement in 39%, 24%, and 16% of their patches that affect the data-flow, respectively. Every fifth patch that affects the data-flow (21%) actually adds a new function call, for instance to report an error or to release resources. A completely new variable is declared in every sixth patch that affects the control flow (14%). Only 2.8% of all patches introduce complex functions that would need to be synthesized.

Patch Complexity. Mutation testing [30] is based on the assumption that test case finding simple errors that are artificially injected are also effective in finding more complex errors. Errors are injected using simple changes for instance by deleting a statement, changing an arithmetic or binary operator, or substituting variables and constants. These ideas have also been applied to automated program repair [31]. However, in our study, we find that only every ninth patch (12%) actually changes an arithmetic or binary operator. Every fifth patch (17%) substitutes a variable or constant by another. Most patches affect only one statement (median 1). Yet, on average, every patch applies about two changes to modify either data-flow or control-flow.

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

E. Implications

We distinguish between *participants* and *respondents* to differentiate between findings from the observational and the retrospective study, respectively.

Automated Documentation. Most respondents frequently debug programs written by other developers. Participants rate errors as very difficult to diagnose often because certain flags, functions, or data structures are left undocumented. The lack of documentation has a detrimental impact on the likelihood to produce a correct patch. Thus, techniques that can summarize and explain the meaning of a function or variable provide substantial benefits. We also present some evidence that tools which assign the employee with the best understanding of the buggy component for debugging can increase the likelihood to patch the error correctly.

Automated Fault Localization. We present more evidence that the assumption of perfect bug understanding is invalid [3], [2], [32]. It is *not* sufficient to point out one suspicious statement in the source code to understand the root cause of an error. Most narratives of the chain of events leading up to the error reference 3–4 code regions that can be distributed across several files. We cannot expect that pointing out a suspicious statement in only one region is sufficient for an effective bug diagnosis. Constructively, we suggest instead to measure the precision of finding *at least one statement in each region that – a developer – identifies* as pertinent even without a tool.

Automated Diagnosis. Many respondents (40%) expressed explicit interest in tools that produce a more sophisticated bug diagnosis than fault localization. We find that most participants (85%) provide basically the same explanation for how an error comes about. So, in principle, a tool could create an agreeable and precise explanation of the pertinent sequence of events leading up to the error. In fact, we aggregate the provided explanations to produce such a narrative. Constructively, we suggest that an effective automated bug diagnosis tool *points to the same variable values, function calls, and data flows that a developer identifies* as pertinent even without a tool.

Patches as Diagnosis. Several respondents (13%) would design an automated repair tool which allows to inspect the generated patches to gain insights about the cause of the error. While diagnoses may span several functions, patches are mostly local to a function. Three of the four errors that are perceived to be very difficult to diagnose are actually caused by an operator fault and can be fixed in a single line. However, we note that only about two-third of patches actually modify statements that are referenced in the bug diagnosis. We suggest to investigate whether patches can serve as adequate diagnosis.

Automated Patch Review. For the median error, *31%* of submitted patches passes the test case but fails the code review. In automated repair research, such patches are considered plausible but incorrect [29]. Even though many participants generate new test cases or execute existing ones, the most important causes of patch incorrectness are regression and incomplete fix. We note that these causes can be addressed with existing techniques, like regression test generation [33], [34] or regression verification techniques [35], [36]. However, every fifth incorrect patch actually treats the symptom, for instance, by removing the failing assertion. Our benchmark provides incorrect patches and reasons as to why they are incorrect. Constructively, we suggest that an effective automated code review tool *detects at least the same incorrect patches* that failed the human patch review.

Automated Program Repair. Many developers provide plausible but incorrect patches. This clearly motivates the need for automated tools that assist in generating a correct patch. We find that there are several ways to patch an error correctly, syntactically and semantically. For instance, to patch an array access out-of-bounds, some participants might increase the initially allocated memory, others might prevent the access, and others might re-allocate memory as necessary. One third of patches exclusively affect the control-flow in some manner. Such patches may be efficiently generated by automated repair techniques such as SPR [37]. Only very few patches would require the synthesis of complex functions. However, many patches actually add a new statement, such as a function call to release resources. Many changes in a patch may not be brought about by simple mutation operators.

V. DBGBENCH: AUTOMATED DEBUGGING BENCHMARK

Our findings suggest to evaluate automated debugging techniques with respect to manual debugging techniques: Does the automated technique report pertinent fault locations, variable values, function parameters, or the sequence of events that a developer finds relevant to point out herself?

We introduce DBGBENCH which consists of 27 errors that the developers introduced in 13 revisions of 2 well-known, well-maintained, and widely-deployed open source C projects, `findutils` and `grep`, taken from COREBENCH [2]. For each error, we provide a failing test case, a simplified bug report, the identified fault locations, an explanation of the runtime actions leading to the error, the time taken to understand and fix the error, and examples of correct and incorrect patches. An overview can be found in the appendix.

Test & Report. For each error there exists at least one *executable, failing test case* and a simplified bug report that contains concrete instructions on how to reproduce the bug, the actual and expected output. For example, the *simplified bug report* for `find.66c536bb` reads:

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.
```

The bug report clearly explains how to reproduce the bug, which symptoms we observe, and which output we expect. Note that this provides the strongest oracle (cf. [29]).

Bug Diagnosis. For each error, we asked our participants to provide the root cause of the error and explain the runtime actions leading to the error while referencing the pertinent locations in the source code. Subsequently, we aggregated these explanations into a single, self-containing, and precise *bug diagnosis*. We note that 15% of participants provide an explanation that does not agree with our diagnosis while rating their confidence in the correctness of their explanation on average only as *slightly confident* (2.4) on a 5-point Likert scale. In contrast, participants with an explanation that agrees with our diagnosis are *very confident* (3.7) in the correctness of their explanation, on average. The aggregated bug diagnosis for `find.66c536bb` reads:

> If `find` is set to print files that are strictly younger than 2 days (-mtime -2), it will instead print files that are exactly 2 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 '2' such that it is incorrectly classified as COMP_EQ (`parser.c:3178`).

Automated debugging tools are expected to reference the same pertinent locations, variables, functions, or chain of events that are provided by our explanation.

Correct Patches. After explaining the root cause of the error, we asked our participants to fix the error and submit the patch. 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. Our benchmark provides several examples of correct patches and a high-level description of the changes done to the code. For example, the error `find.66c536bb` can be patched correctly as follows:

1) Copy timearg and restore after first call to get_comp_type.
2) Pass a copy of timearg into first call of get_comp_type.
3) Pass a copy of timearg into call of get_relative_timestamp.
4) Decrement timearg after the first call to get_comp_type.

Incorrect Patches. For each incorrect patch we give a reason as to why it is incorrect and whether the test case passes. Reasons are listed in Figure 7. The error `find.66c536bb` has the following example for an incorrect patch:

> Restore timearg only if classified as COMP_LT (*Incomplete Fix* because it does not solve the problem for -mtime +2).

Usage. DBGBENCH (see appendix) allows to evaluate novel automated debugging and patching techniques and assistants:

- The human-generated fault locations can be used to evaluate *automated fault localization* techniques. We suggest to measure the accuracy in finding at least one statement in each contiguous region that participants localized.
- The human-generated explanations can be used to evaluate *automated bug diagnosis* techniques. We suggest to measure the accuracy in finding the pertinent variable values, function calls, events, or cause-effect chains mentioned in the aggregated human-generated bug diagnosis.
- The examples of correct and incorrect patches can be used to evaluate *automated repair and code review* techniques. These high-level explanations serve as ground-truth to determine the correctness (not plausibility) of an auto-generated patch.
- The time that our participants take to understand and patch each error can be used to measure *how much faster developers can be if assisted with automated tools*.

VI. Limitations

In software engineering, it is often difficult to draw general conclusion from empirical studies because a potentially large number of contextual variables can impact the process under investigation [38]. Since we investigate the debugging of just two programs, we cannot assume generalization of the findings of the observational study. 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. However, there is nothing specific to our investigations that would prevent replication for errors in other programs. In fact, *we strongly urge the community to reproduce our study* for different programming languages and applications domains to build an empirical body of knowledge, to establish the means of evaluating automated debugging techniques more faithfully. DBGBENCH is the first human-generated benchmark for the evaluation of automated diagnosis and repair techniques and may serve as subject for *in-depth case studies*. To facilitate replication, the questionnaires for the retrospective and observational study are made available [39].

In empirical research, in-depth case studies that involve only two subjects 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" [40]. This does not mean that research focusing on large samples is not important. On the contrary, both types of research are essential [41].

As potential threat to internal validity, we note that we suggested 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. In order 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".

VII. Conclusion and Consequences

Despite the surge in publications on automated debugging in the past decade, debugging is still an under-researched field—maybe not so much how tools may find faults, but more how practitioners actually debug programs, and how approaches may address their needs and processes. The DBGBENCH benchmark, introduced in this paper, provides essential data to guide future research in the field, and gives insights into the large variance at which practitioners diagnose and repair faults. DBGBENCH is available at the project website:

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

The initial analysis of our studies, as presented in this paper, only scratches the surface of what can be done with the DBGBENCH benchmark. Our future work will focus on the following topics:

find.24e2271e
Error Type: Functional Bug
Avg. Time: 13.8 min
Explanation: Slightly difficult
Patching: Slightly difficult
Correctness: 75%

If find is set to print the found file's base directory followed by the found file's name (-printf '%H %P\n') and there exist directories of different length, then find incorrectly splits base directory and file name during printing. Because the index state.starting_path is set only for the first working directory (`ftsfind.c:278-279`) the incorrect value of state.starting_path is used when printing base directory and file name (`pred.c:709-718`, `pred.c:813`). **Examples of Correct Fixes**: 1) Recompute state.starting_path_length for each argv before calling find. 2) Weaken condition that prevents state.starting_path_length to be reset. **Example of Incorrect Fix**: Always update state.starting_path_length even if ent->fts_level != 0 (*Regression* because it then carries the incorrect "starting_path_length").

find.dbcb10e9
Error Type: Crash
Avg. Time: 22.9 min
Explanation: Slightly difficult
Patching: Slightly difficult
Correctness: 81%

If find is set to print all files that are exactly 2 days old (-mtime 2), it crashes with a segmentation fault. Variable **pend is defined as pointer pointer (`parser.c:2739`) and expected to be allocated when xstrtoumax is called (`parser.c:2759`). However, it is still NULL after the call such that the null pointer check for pend* is itself a null pointer dereference (`parser.c:2762`). **Examples of Correct Fixes**: 1) Add null pointer check for pend. 2) Change definition of **pend to *pend and update references. 3) Allocate memory for **pend. **Examples of Incorrect Fixes**: 1) Remove code containing null pointer dereference (*Treating the Symptom*). 2) Change the check involving **pend (*Treating the Symptom* because the nullpointer is still dereferenced, only the program does not crash).

find.07b941b1
Error Type: Crash
Avg. Time: 23.7 min
Explanation: Slightly difficult
Patching: Slightly difficult
Correctness: 80%

If find is set to search for file matching a regular expression (-regex '.*'), the argument pointer arg_ptr is incremented (`parser.c:1644`) before it is used (`parser.c:1645`) which results in a nullpointer dereference (`parser.c:926`). **Examples of Correct Fixes**: 1) Increment arg_ptr *after* argv[*arg_ptr] is read. 2) Save the previous value of argv[*arg_ptr] in a temporary variable and use this one. **Examples of Incorrect Fixes**: 1) Do not increment arg_ptr at all (*Regression* because other arguments may not be parsed at all). 2) Add null-pointer check (*Incomplete Fix* because estimate_pattern_match is still called with a nullpointer).

find.c8491c11
Error Type: Crash
Avg. Time: 31.4 min
Explanation: Slightly difficult
Patching: Slightly difficult
Correctness: 54%

If find is set to print files that are newer than a reference file and this reference file is not specified (-newerXY), find crashes with a segmentation fault. This is caused by incrementing the argument pointer arg_ptr without a bounds check (`parser.c:1315`) resulting in a null pointer dereference (`lib/quotearg.c:249`). **Examples of Correct Fixes**: Check for nullpointer directly after increment of arg_ptr. **Examples of Incorrect Fixes**: 1) Check for nullpointer only before or in fatal_file_error (*Incomplete Fix* because null pointer might still propagate via parser.c:1342 or parser.c:1347). 2) Do not increment the pointer at all (*Regression* because some arguments may not be parsed, at all).

find.6e4cecb6
Error Type: Functional Bug
Avg. Time: 38.2 min
Explanation: Moderately difficult
Patching: Not at all difficult
Correctness: 89%

If find is set to search a directory referenced by a symbolic link and containing a file, and find is set to follow symbolic links (-L) or to not follow symbolic links except for those set to be searched (-H), then find does not print the file in the referenced directory and instead reports "Too many levels of symbolic links". Because of a mixup in the condition of a ternary operator (`find.c:1094`), extraflags are set to O_NOFOLLOW when it should be 0 and to 0 when it should be O_NOFOLLOW. The flag controls whether symlinks are followed when a directory is opened (`find.c:1097`). Because of this fault, safely_chdir returns SafeChdirFailSymlink (`find.c:1618`) whence the error message is printed (`find.c:1642`). **Example of Correct Fix**: Fix ternary operator. **Example of Incorrect Fix**: Do not fail if safely_chdir returns SafeChdirFailSymlink (*Treating the Symptom*).

find.091557f6
Error Type: Crash
Avg. Time: 44.8 min
Explanation: Slightly difficult
Patching: Slightly difficult
Correctness: 54%

If find is set to search for files (-type f) while following symbolic links (-L) and a symbolic link loop exists, then it aborts with a coredump instead of listing the symbolic links and terminating gracefully. If a symbolic link loop exists, no stat information is available and the flag FTS_NS is set (`ftsfind:584`). The flag is not properly handled (`ftsfind.c:425-446`), such that state.type and mode are incorrectly set (`ftsfind.c:460`) and the assertion fails (`pred.c:1578`). **Example of Correct Fix**: Handle FTS_NS flag. **Examples of Incorrect Fixes**: 1) Remove violated assertion (*Treating the Symptom*). 2) Force stat() to be called such that stat information is available (*Incorrect Workaround* because stat() is not supposed to be called on symlink loops).

find.24bf33c0
Error Type: Crash
Avg. Time: 45.1 min
Explanation: Moderately difficult
Patching: Slightly difficult
Correctness: 50%

If find is set to search for files (-type f) while following symbolic links (-L) and a symbolic link loop exists, then it still prints the looping links while an error message is expected. If a symbolic link loop exists, no stat information is available and the flag FTS_NS is set (`ftsfind.c:586`). The flag is not properly handled (`ftsfind.c:431-446`) so that the links are printed (`pred.c:1459`). **Example of Correct Fix**: Handle FTS_NS as error IF symlink loop. **Examples of Incorrect Fixes**: 1) Handle FTS_NS as error independent of whether it is a symlink loop (*Regression* because FTS_NS alone does not indicate an error). 2) Handle all flags as error (*Regression* because not all flags indicate errors).

find.183115d0
Error Type: Resource Leak
Avg. Time: 49.2 min
Explanation: Slightly difficult
Patching: Slightly difficult
Correctness: 83%

If we ulimit the number file descriptors that can be open simulatanously and set find to execute ls for every subdirectory (-execdir ls '{}' \;), it quickly runs out of file descriptors. File descriptors are always opened (`pred.c:520`) but never closed (`pred.c:659-664`) which raises an error when no more descriptors are available (`pred.c:579`). **Example of Correct Fix**: Close file descriptor as soon as it is not used anymore. **Example of Incorrect Fix**: Close random file descriptor (*Incomplete Fix* because still leaking file descriptors).

find.93623752
Error Type: Functional Bug
Avg. Time: 50.8 min
Explanation: Moderately difficult
Patching: Slightly difficult
Correctness: 92%

There are two errors: 1) If find is set to search for files that were changed in the last n days but n is not a number (-ctime x), then find complains about a "missing" argument instead of reporting the "incorrect" argument. Function parse_time calls collect_args to assign the current argument argv[*arg_ptr] to timearg and increment the argument pointer arg_ptr (`parser.c:3102`). When timearg is failed to be parsed as a number, parse_time returns without decrementing arg_ptr (`parser.c:3127-3128`). When the error is reported (`tree.c:1248-1271`), the argument pointer points to NULL directly after the incorrect argument (`tree.c:1250`), such that the error is reported as missing argument instead of invalid argument. 2) If find is set to search for files belonging to a certain group but the group-id is not specified or not a number (-gid x), then find crashes with a segmentation fault. When the argument following the -gid option is being parsed (`parser.c:913`), insert_num returns NULL because argv[*arg_ptr] is NULL or not a number (`parser.c:3235-3259`). This nullpointer remains unchecked and is dereferenced leading to a segmentation fault (`parser.c:914`). When nullpointer dereference is fixed the same symptom is observed for -gid as for -ctime because the argument pointer is also forgot to be decremented. **Examples of Correct Fixes**: For first error, 1) decrement/restore arg_ptr when parsing of second argument of an option fails or 2) use copy of old argument during error-reporting. For second error, add null pointer check. **Example of Incorrect Fix**: For first error, decrement argument pointer before even calling parse_time (*Regression* because even correct arguments are reported as incorrect ones).

find.66c536bb
Error Type: Functional Bug
Avg. Time: 55.5 min
Explanation: Moderately difficult
Patching: Slightly difficult
Correctness: 92%

If find is set to print files that are strictly younger than 2 days (-mtime -2), it will instead print files that are exactly 2 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 '2' such that it is incorrectly classified as COMP_EQ (`parser.c:3178`). **Examples of Correct Fixes**: 1) Save timearg in auxiliary variable and restore after first call to get_comp_type. 2) Pass a copy of timearg into the first call of get_comp_type. 3) Pass a copy of timearg into get_relative_timestamp (which calls get_comp_type the second time). 4) Decrement timearg after the first call to get_comp_type. **Example of Incorrect Fix**: Restore timearg only if classified as COMP_LT (*Incomplete Fix* because it does not solve the problem for -mtime +2).

find.b445af98
Error Type: Functional Bug
Avg. Time: 56.5 min
Explanation: Moderately difficult
Patching: Slightly difficult
Correctness: 50%

If find is set to search a directory containing a symbolic link, to not follow any symbolic links (except for those specified on the command line; -H), and to print only symbolic links (-type l), then find does not print the link. The root cause is that state.cur_depth is used before it is set. When digest_mode checks whether to follow symlinks (`util.c:629`), state.curdepth is still 0 (`util.c:607`), so that mode are incorrectly set to follow symlinks (`util.c:630-636`). Only later state.curdepth is set (`ftsfind.c:230`). Because of the incorrect value of mode, it is incorrectly decided not to print the file (`pred.c:1749`). **Example of Correct Fix**: Move state.curdepth assignment to shortly before digest_mode is called. **Examples of Incorrect Fixes**: 1) Change check to match incorrect value (0) of state.curdepth (*Treating the Symptom*). 2) Force stat() to be called such that stat information is available (*Incorrect Workaround* because stat() is not supposed to be called on symlink loops).

find.ff248a20
Error Type: Infinite Loop
Avg. Time: 57.7 min
Explanation: Moderately difficult
Patching: Moderately difficult
Correctness: 40%

If find is set to search a directory containing a symbolic link that references an ancistor directory and if find is set to follow symlinks (-follow), then it runs indefinitely. The global variable dir_ids tracks the directories that have already been visited. The function process_path would correctly exit with a loop warning (`find.c:1428-1434`) if the current directory (in stat_buf) has already been visited. However, after the current directory is correctly added to those that have already been visited (`find.c:1442`), the same entry is overriden with uninitialized values (`find.c:1621`) such that the current directory is never marked as already visited. **Examples of Correct Fixes**: 1) Remember whether stat() has been called. If not done, call stat() before overriding dir_ids[dir_curr] at find/find.c:1621. 2) Always stat() before overriding dir_ids[dir_curr] at find/find.c:1621 such that statbuf is initialized. 3) Only overwrite dir_ids[dir_curr] if statbuf is initialized. **Examples of Incorrect Fixes**: 1) Never override dir_ids[dir_curr] (*Regression* because it isn't overridden when it should be). 2) Follow links to a maximum depth of 1 (*Regression* because symlinks might need to be followed to an arbitrary depth).

find.e6680237
Error Type: Functional Bug
Avg. Time: 76.4 min
Explanation: Moderately difficult
Patching: Moderately difficult
Correctness: 27%

If find is set to search a directory containing three other directories which contain the folder "bug" and to execute pwd in every folder containing the folder "bug" (-name bug -execdir pwd \;), then find prints the first directory three times. The reason is that the working directory specified in execp->wd_for_exec is set only once (`pred.c:513-527`) and never updated. **Examples of Correct Fixes**: 1) Correct buggy if-condition by substituting excep->wd_for_exec by execp->todo. 2) If is_exec_in_local_dir, then always reallocate execp->wd_for_exec and remove the assertion. **Example of Incorrect Fix**: Remove if-condition such that it always redefines execp->wd_for_exec and keep assertion that execp->todo is false (*Regression* because execp->todo might be true such that assertion may fail).

find.e1d0a991
Error Type: Functional Bug
Avg. Time: 88.2 min
Explanation: Very difficult
Patching: Very difficult
Correctness: 17%

If find is set to a directory containing a file, to follow symbolic links (-L), and to execute ls for every subdirectory (-execdir ls '{}' \;), then find incorrectly also prints the base directory. If find is set to follow symlinks, the flag FTS_LOGICAL is set (`ftsfind.c:349`) before the directory search is initiated (`ftsfind.c:364`). When a directory is searched (`ftsfind.c:373`), the working directory is not changed because FTS_LOGICAL is set. Hence, the *full* pathname is passed as argument to execdir (`pred.c:484-490` and `pred.c:467-471`). **Example of Correct Fix**: Correctly compute pathname and prefix in new_impl_pred_exec. **Example of Incorrect Fix**: Remove FTS_LOGICAL flag (Incorrect Workaround because FTS_LOGICAL is supposed to be set).

grep.55cf7b6a
Error Type: Functional Bug
Avg. Time: 21.1 min
Explanation: Slightly difficult
Patching: Not at all difficult
Correctness: 91%

If grep is set to silently skip devices, FIFOs, and sockets (-D skip), then grep does not search on standard input when no file is provided. When the skip option is enabled, variables devices is set to SKIP_DEVICES (`main.c:1852-1859`). If no file is provided, variable file is NULL and variable desc is set to STDIN_FILENO (`main.c:1217-1218`). The code which handles SKIP_DEVICES (`main.c:1246-1255`) decides to skip STDIN (which is a special device) even though it should not (desc == STDIN_FILENO). **Examples of Correct Fixes**: 1) Do not skip if desc is set to STDIN_FILENO. 2) Do not skip if file is not set (and thus desc is set to STDIN_FILENO). **Example of Incorrect Fix**: Negate the skip condition (*Regression* because it skips everything that should not be skipped while indeed not skipping STDIN).

grep.54d55bba
Error Type: Crash
Avg. Time: 26.7 min
Explanation: Slightly difficult
Patching: Slightly difficult
Correctness: 69%

If grep is set to search in all files under each directory recursively (-r) but to exclude certain directories (–exclude-dir=foo), then grep crashes with a segmentation fault. When grepdir computes the name_space (`src/grep.c:1361`), it calls function isdir1 via function savedir (`lib/savedir.c:123`). Now, the code in isdir1 that is supposed to remove the trailing slashes from the directory name uses the uninitialized variable path instead of variable dir (`lib/savedir.c:51`). The nullpointer dereference results in a segmentation fault. **Example of Correct Fix**: Substitute path with dir. **Examples of Incorrect Fixes**: 1) Return if path is not initialized (*Regression* because isdir1 returns false even if dir is a directory). 2) Only use path if initialized (*Regression* because isdir1 does not remove trailing slashes).

grep.9c45c193
Error Type: Functional Bug
Avg. Time: 37.7 min
Explanation: Moderately difficult
Patching: Slightly difficult
Correctness: 83%

If grep is set to search only specific files (–include=a.txt), then grep does not print a match even if there is one. First, main correctly adds the include pattern with EXCLUDE_INCLUDE flag set (`grep.c:2136-2140`). When the files are chosen for the search, files that are supposed to be included are actually excluded because the return value of excluded_file_name is unnecessarily negated (`grep.c:2267-2269`). The negation is unnecessary because the function exclude_file_name is incorrectly assumed to treat excludes and includes the same. However, the behavior changes if the EXCLUDE_INCLUDE flag is present (`lib/exclude.c:410`, `lib/exclude.c:359`). **Examples of Correct Fixes**: 1) Remove negation such that included_patterns are not excluded during classification. 2) Do not set EXCLUDE_INCLUDE flag for included_patterns which effectively negates the faulty condition. **Example of Incorrect Fix**: Independent of whether a file matches the included_pattern, never exclude (*Regression* because it doesn't skip files that are *not* in the included patterns).

grep.5fa8c7c9
Error Type: Infinite Loop
Avg. Time: 38.8 min
Explanation: Moderately difficult
Patching: Slightly difficult
Correctness: 50%

If grep is set to search for fixed strings (-F), the empty string is given (""), and the locale is UTF8, then grep runs undefinitely. When FExecute searches for a match of the empty string, variable len contains the size of the match; here, len=0 (`kwsearch.c:106`). Because len=0, the check is_mb_middle (`searchutils.c:117-146`) whether the match occurs within a multibyte character returns true (`kwsearch.c:108`). However, the size of the supposed multibyte character is computed as mb_len=1 (`kwsearch.c:115`). When mb_len-1 is added to beg (`kwsearch.c:118`) to advance behind the supposed multibyte character, beg's value remains unchanged. The loop is continue'd (`kwsearch.c:121`). Since beg has the same value every time the loop exit condition is checked (`kwsearch.c:101`), the loop exit condition never holds, resulting in an infinite loop. **Examples of Correct Fixes**: 1) Function is_mb_middle returns false for len=0. 2) Only call is_mb_middle if len is set. 3) Jump to success if mb_len==1. **Examples of Incorrect Fixes**: 1) Remove continue (*Treating the Symptom*). 2) Don't reset beg (*Regression* because it breaks multibyte character handling). 3) Remove part of the check which causes is_mb_middle to return true (*Regression* because it breaks multibyte character handling). 4) Do not compute match_size but teturn complete buffer until end of line (*Regression* because only match should be returned).

grep.db9d6340
Error Type: Infinite Loop
Avg. Time: 40.6 min
Explanation: Slightly difficult
Patching: Slightly difficult
Correctness: 45%

If grep conducts a fixed-strings search (-F) for a pattern that contains multibyte characters, then it runs indefinitely. When EXECUTE_FCT finds a match in the middle of a multibyte character, it is supposed to continue after the multibyte character (`search.c:638-639`). However, the beginning of the next multibyte character is not found, and mb_start remains unchanged (`search.c:228-256`). After beg is assigned mb_start minus 1, the loop is continue'd (`search.c:640`). The loop exit condition never holds (`search.c:632`) because beg never exceeds buf + size, resulting in an infinite loop. **Examples of Correct Fixes**: 1) Raise an error, if is_mb_middle is unsuccessful in finding the beginning of the multi-byte and adjusting mb_start. 2) Go to after the current match. **Examples of Incorrect Fixes**: 1) Remove continue (*Treating the Symptom*). 2) Do not reset beg (*Regression* because it breaks multibyte character handling).

grep.2be0c659
Error Type: Functional Bug
Avg. Time: 47.2 min
Explanation: Moderately difficult
Patching: Moderately difficult
Correctness: 13%

If grep conducts a case-insensitive search (-i) in a file containing 8-bit characters and the current locale is Turkish UTF8, then grep prints the wrong output. When grep conducts a case-insensitive search, it lowers the case of the input string before matching (`search.c:384-392`). The lower case of an upper-case 8-bit character might occupy one more or less bytes. The latter case is not handled. When the match_size is computed (`grep.c:1081`), the lower-case match is used (`grep.c:1060-1062`). When the match is printed, the incorrect lower-case match size which is usually larger than the actual match size is used (`grep.c:1085-1091`). **Examples of Correct Fixes**: 1) Update the map that maps lower-case character to the normal case characters to account for cases where the number of bytes it occupies *decreases* in the lower-case. 2) To correct the match_size, lower-case as many characters in the normal-case match as result in match_size lower-case characters. **Examples of Incorrect Fixes**: 1) Return complete line if match exists (*Regression* because only the match should be returned). 2) Add the difference in length of lower-case and normal-case string to the match size (*Incomplete Fix* because for files that have more multibyte characters than given in the match, grep reports longer matches than needed).

grep.8f08d8e2
Error Type: Functional Bug
Avg. Time: 48.4 min
Explanation: Moderately difficult
Patching: Moderately difficult
Correctness: 75%

If grep is set to search for lines containing whole words that match a regular expression (-w), it prints only the match instead of the complete line. When execute searches for a match, it correctly sets variable len to the length of the match (`search.c:388`). When it is checked if the match aligns with word bounderies (`search.c:408-414`), the match length len still points to the end of the match. So, execute returns the length of the match instead of the end of the line (`grep.c:997`). **Examples of Correct Fixes**: 1) Add statement: goto success (which updates len with end - beg). 2) Update len with end - beg. **Example of Incorrect Fix**: Always return complete line (*Regression* because in some settings grep should return only the match).

grep.58195fab
Error Type: Functional Bug
Avg. Time: 50.5 min
Explanation: Moderately difficult
Patching: Slightly difficult
Correctness: 82%

If grep is set to search all TXT files (–include="*.txt") but excluding some files (–exclude="foo.txt"), then grep also searches files that are not TXT ignoring the include option. Because included_patterns is not initialized with EXCLUDE_WILDCARDS (`src/grep.c:2137`), the exclude pattern is not added in add_exclude (`lib/exclude.c:449`). Files are matched exactly (treating "*.txt" as file name) instead of using wildcards (`lib/exclude.c:417-427`). These files are then incorrectly classified as included/excluded (`src/grep.c:2261-2271`). **Examples of Correct Fixes**: 1) Add EXCLUDE_WILDCARDS flag for includes. 2) Add EXCLUDE_INCLUDE flags for excludes if there are includes. **Examples of Incorrect Fixes**: 1) Substitute EXCLUDE_INCLUDE with EXCLUDE_WILDCARDS for includes (*Regression* because EXCLUDE_INCLUDE flags must also be set for includes). 2) Negate condition that decides whether to exclude (*Regression* because files that are specified to be excluded are now included).

grep.c1cb19fe
Error Type: Functional Bug
Avg. Time: 58.4 min
Explanation: Very difficult
Patching: Slightly difficult
Correctness: 71%

If grep searches for string specified in a bracket expression, then for some UTF8 locales (ru_RU.UTF-8) grep does not print a match. For some locales dfaparse sets the global flag hard_LC_COLLATE (`dfa.c:1418`) to denote that characters are ordered in a strange way (e.g. Russian cyrilic). If hard_LC_COLLATE is set, then lex prepares the info about the letters in the bracket expression and finally calls in_coll_range (`dfa.c:1103-1116`). Now, in_coll_range uses the correct function strcoll to compare the letters, but the condition is incorrect and the wrong character are selected to be in the range that is specified by the bracket expression. Hence, there is no match reported. **Example of Correct Fix**: Fix the simple operator fault. **Examples of Incorrect Fixes**: 1) Fix locale, such that multibyte characters do not need to be handled (*Regression* because LC_ALL is supposed to be handled). 2) Implement in_coll_range as locale implemented match (*Regression* because match is supposed to be locale dependent).

grep.7aa698d3
Error Type: Functional Bug
Avg. Time: 59.9 min
Explanation: Moderately difficult
Patching: Moderately difficult
Correctness: 13%

If grep conducts a case-insensitive search (-i) on an input that contains multibyte characters and the locale is UTF8, then grep prints a match of incorrect length. When conducting the case-insensitive search, EXECUTE_FCT first computes a lower-case of the input (`search.c:388`). The length of the match is computed for the match in the lower-case input (`search.c:555`). However, the lower-case of a multibyte character can take 1 byte less. So, the length of the normal-case and lower-case input differ. The computed value of match_size could be half the expected value (`grep.c:1081-1085`). Hence, the match in the normal-case input is printed with incorrect length (`grep.c:1091`). **Example of Correct Fix**: Add a mapping between normal-case and lower-case string to compute the length of the match in the normal-case string from the length of the match in the lower-case string. **Examples of Incorrect Fixes**: 1) Do not lower the case (*Regression* because a case-insensitive search is case-sensitive). 2) If matched string contains a multibyte char, double the match_size (*Incomplete Fix* because it works only of all are multibyte characters). 3) Print complete line if there is a match (*Regression* because only match should be returned).

grep.3220317a
Error Type: Crash
Avg. Time: 63.7 min
Explanation: Moderately difficult
Patching: Moderately difficult
Correctness: 20%

If grep searches for a bracket expression containing a multibyte character in a file that contains multibyte characters and the current locale is UTF8, then grep crashes with a segmentation fault. When parse_bracket_exp parses the next character, array index c is assigned EOF (-1) if the character is multibyte (`dfa.c:498`, `dfa.c:363`) while wc is assigned the correct index. However, when parse_bracket_exp calls setbit_case_fold (`dfa.c:697`) it uses c which overflows during the cast from int to unsigned. After setbit_case_fold has called setbit (`dfa.c:274`), the array is accessed at a too large index which causes a segmentation fault (`dfa.c:168`). **Example of Correct Fix**: Use wc instead of c (which equals c if the character is not multibyte). **Examples of Incorrect Fixes**: 1) Check for overflow condition c=EOF (*Treating the Symptom* because multibyte characters are still handled incorrectly). 2) Use an arbitrary value instead of c (*Treating the Symptom* because while it does not crash, the bracket expression is not correctly handled).

grep.3c3bdace
Error Type: Crash
Avg. Time: 64.8 min
Explanation: Very difficult
Patching: Moderately difficult
Correctness: 70%

If grep searches for a certain extended regular expression (-E '(^|)*(| $)'), then it crashes with a coredump. When dfaanalyze allocates memory for merged.elems (`dfa.c:1728`), it allocates insufficient memory because merged.elems can grow to twice the original size (`dfa.c:1455`). Then memory is corrupted when the array is accessed out of bounds (`dfa.c:1453`). Only later the program crashes because of the corrupted memory (`dfa.c:1917`). **Examples of Correct Fixes**: 1) Allocate twice or 3x as much for merged.elems. 2) Reallocate as needed. **Example of Incorrect Fix**: Always reset the number of elements (nelem) to 0 (*Regression* because we always override the first element).

grep.c96b0f2c
Error Type: Functional Bug
Avg. Time: 67.6 min
Explanation: Very difficult
Patching: Moderately difficult
Correctness: 50%

If grep conducts a case-insensitive search (-i) for the empty line ('^$') and an UTF-8 locale is set, then grep reports matches even for non-empty lines. For case-sensitive searches or 8-bit locales, execute is called with the complete buffer and correctly returns no match (`grep.c:1045-1046`). Otherwise, execute is called for each line (`grep.c:1048-1063`). However, execute does not handle the case when no match is found (`search.c:388`), which is why the non-match is printed (`grep.c:1091`). **Examples of Correct Fixes**: 1) Handle case where no match was found by breaking loop if next_beg == buflim. 2) Skip printing if match is empty and we are not in inversion mode (-v). **Example of Incorrect Fix**: Skip printing if match is empty even if in inversion mode (*Regression* because it breaks inversion mode).

Fig. 9. Complete list of errors and their average debugging time, difficulty, and patch correctness, with human-generated explanations of the runtime actions leading to the error, and examples of correct and incorrect fixes, sorted according to average debugging time (zoom required).

Better Diagnosis Tools. From both our studies, it became clear that automatically predicting a location (or a set of locations) does not provide sufficient support for developers. Descriptions that describe the circumstances of the error and the cause-effect chain of how it came to be (including associated variables and locations) would likely be much more helpful; but while humans can easily narrate these (Figure 9), producing these from automated tools is still a long way to go.

Better Repair Tools. Given the several incorrect or incomplete fixes we found in our observational study, it is evident that much better support for repairs is needed. Tools and approaches that validate a repair for correctness, determine whether a repair addresses the cause or a symptom, and can choose between multiple repairs would certainly be appreciated. Strong automated support for repairing bugs might require much better tests or specifications, as experience with automated repair tools suggests [42].

Expectations on Automatic Debugging Tools. In our retrospective study, we also asked respondents about properties that make an automatic patch acceptable, as well as additional expectations on automated bug diagnosis tools; these answers and their consequences will be discussed in an extended version of this paper.

ACKNOWLEDGMENTS

The authors would like to thank all participants and respondents for their commitment towards this project and their dedication in fixing even the hardest bugs. Curd Becker of CISPA/Saarland University provided valuable technical support throughout the project.

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. 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.

[3] 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.

[4] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *ISSTA*, 2016, pp. 165–176.

[5] 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.

[6] P. Romero, B. du Boulay, R. Cox, R. Lutz, and S. Bryant, "Debugging strategies and tactics in a multi-representation software environment," *International Journal of Human-Computer Studies*, vol. 65, no. 12, pp. 992 – 1009, 2007.

[7] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, Feb 2013.

[8] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

[9] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR*, 2005, pp. 1–5.

[10] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, 2007.

[11] R. Likert, "A technique for the measurement of attitudes." *Archives of psychology*, 1932.

[12] W. Hudson, *Card Sorting*, 2013.

[13] P. J. Guo, "Online Python tutor: embeddable web-based program visualization for cs education," in *SIGCSE*, 2013, pp. 579–584.

[14] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10, 2010, pp. 43–52.

[15] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.

[16] V. Raychev, M. T. Vechev, and A. Krause, "Predicting program properties from "big code"," in *POPL*, 2015, pp. 111–124.

[17] J. A. Clause and A. Orso, "Penumbra: automatically identifying failure-relevant inputs using dynamic tainting," in *ISSTA*, 2009, pp. 249–260.

[18] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: automated synthesis of repair hints," in *ICSE*, 2014, pp. 266–276.

[19] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002, pp. 1–10.

[20] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken, "Statistical debugging of sampled programs," in *NIPS*, 2003, pp. 603–610.

[21] 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.

[22] "Valgrind instrumentation framework," http://valgrind.org.

[23] "Observational study infrastructure," https://drive.google.com/open?id=0Bx6dkN27OssKVWJYZGdXcWdWQ0U, accessed: 2016-08-22.

[24] I. R. Katz and J. R. Anderson, "Debugging: An analysis of bug-location strategies," *Hum.-Comput. Interact.*, vol. 3, no. 4, pp. 351–399, Dec. 1987.

[25] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002.

[26] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in Java programs," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2016.

[27] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 254–265.

[28] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012.

[29] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, 2015, pp. 24–36.

[30] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[31] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *2010 Third International Conference on Software Testing, Verification and Validation*, April 2010, pp. 65–74.

[32] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *ICSE*, 2015, pp. 913–923.

[33] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08, 2008, pp. 218–227.

[34] 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.

[35] ——, "Partition-based regression verification," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 302–311.

[36] B. Godlin and O. Strichman, "Regression verification: proving the equivalence of similarprograms," *Software Testing, Verification and Reliability*, vol. 23, no. 3, pp. 241–258, 2013.

[37] 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.

[38] V. R. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 456–473, Jul 1999.

[39] "DbgBench," http://www.st.cs.uni-saarland.de/debugging, accessed: 2016-08-22.

[40] A. Kuper and J. Kuper, *The Social Science Encyclopedia*. Routledge, 1985.

[41] B. A. Kitchenham and S. L. Pfleeger, *Personal Opinion Surveys*. London: Springer London, 2008, pp. 63–92.

[42] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, May 2014.