How Developers Diagnose and Repair Software Bugs

Marcel Böhme • Ezekiel O. Soremekun • Sudipta Chattopadhyay • Emamurho J. Ugherughe • Andreas Zeller

https://www.st.cs.uni-saarland.de/debugging/dbgbench/
Debugging
A Survey on Software Fault Localization

Abstract:
Software fault localization, the act of identifying the locations of faults in a program, is widely recognized to be one of the most tedious, time consuming, and expensive activities in software development. Due to the increasing costs and complexity of software today,
Are Automated Debugging Techniques Actually Helping Programmers?

Chris Parnin and Alessandro Orso
Georgia Institute of Technology
College of Computing
{chris.parnin|orso}@gatech.edu

ABSTRACT
Debugging is notoriously difficult and extremely time consuming. Researchers have therefore invested a considerable amount of effort in developing automated techniques and tools for supporting various debugging tasks. Although potentially useful, most of these techniques have yet to demonstrate their practical effectiveness. One common limitation of existing approaches, for instance, is their reliance on a set of strong assumptions on how developers behave when debugging (e.g., the fact that examining a faulty statement in isolation is enough for a developer to understand and fix the corresponding bug). In more general terms, most existing techniques just focus on selecting subsets of potentially interesting statements, for instance, by evaluating program slicing [26], only a handful of empirical techniques and whether such use is actually beneficial.

We believe that, given the maturity of the field, it is now time to take into account the inherent complexity of these debugging activities (e.g., the fact that examining a faulty statement in isolation is always enough for a developer to understand and fix the corresponding bug). In more general terms, most existing techniques just focus on selecting subsets of potentially interesting statements, for instance, by evaluating program slicing [26], only a handful of empirical techniques and whether such use is actually beneficial. Such an evaluation of debugging techniques. Such an evaluation should involve studies on how real developers use existing debugging techniques and whether such use is actually beneficial.

We believe that, given the maturity of the field, it is now time to take into account the inherent complexity of these debugging activities (e.g., the fact that examining a faulty statement in isolation is always enough for a developer to understand and fix the corresponding bug). In more general terms, most existing techniques just focus on selecting subsets of potentially interesting statements, for instance, by evaluating program slicing [26], only a handful of empirical techniques and whether such use is actually beneficial. Such an evaluation of debugging techniques. Such an evaluation should involve studies on how real developers use existing debugging techniques and whether such use is actually beneficial.
How Do Developers Debug?

A Survey

An Experiment

A Benchmark
A Survey

- Surveyed developers on
  - time spent on debugging
  - familiarity with debugged code
  - debugging techniques used
  - debugging techniques needed
We distinguish between three debugging tasks:

"Bug Reproduction" Understanding the (user- or auto-generated) bug report and reproducing the bug. Output: Program input that exposes the bug.

"Bug Diagnosis" Understanding the runtime actions leading to the error and identifying the faulty statements in the source code. Output: Explanation of the bug.

"Bug Fixing" Restructuring the faulty source code to remove the error. Output: Fixed program that is at least as correct.

8. How much of your "development time" do you spend reproducing, understanding, and fixing reported bugs? *
Mark only one oval.
- 5% or less
- 5 - 10%
- 10 - 20%
- 20 - 30%
- 30 - 40%
- 40 - 50%
- 50 - 60%
- 60 - 70%
- 70 - 80%
- 80 - 90%
- 90% or more

9. How much of your "debugging time" do you spend with each of the following tasks? *
Mark only one oval per row.

Less than 5%
5 - 10%
10 - 20%
20 - 30%
30 - 40%
40 - 50%
50 - 60%
60 - 70%
70 - 80%
80 - 90%
More than 95%

Bug Reproduction
Bug Diagnosis
Bug Fixing

A Survey

10. When you are debugging, how often is time spent debugging other people’s source code? *
Mark only one oval.
- Never
- Rarely
- Sometimes
- Often
- Always

11. How often do you use the following Bug Diagnosis techniques? *
Mark only one oval per row.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Never</th>
<th>Rarely</th>
<th>Sometimes</th>
<th>Often</th>
<th>Always</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace-based Debugging (using printing; e.g., println, log4c)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interactive or Online Debugging (using breakpoints; e.g., gdb, jdb)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Post-Mortem or Offline Debugging (using core dumps and stack traces)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delta Debugging to minimize failure-inducing input (e.g., AskIgor)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Regression Debugging to identify failure-inducing changes (e.g., git bisect)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statistical or Spectrum-based Debugging to find suspicious statements (e.g., Tarantula)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Slicing (e.g., Frama-C, CodeSurfer)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time Travel or Reversible Debugging (e.g., UndoDB)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Algorithmic or Declarative Debugging (e.g., Java DD)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

12. Are there other "automated" Bug Diagnosis techniques not listed that you use Always or Often? Please specify in one to three words!
Demographics

A Survey

- Advertised on Upwork, Freelancer, Github...
- 180 developers participated
- Majority with 7+ years of experience
- 1/4 students, 1/6 researchers
- Ran over 18 months
Debugging Techniques Used

- Statistical Debugging
- Algorithmic Debugging
- Time Travel Debugging
- Program Slicing
- Regression Debugging
- Post-Mortem Debugging
- Interactive Debugging
- Trace-based Debugging

Frequency:
- Never
- Rarely
- Sometimes
- Often
- Always
What do Developers need?

- Asked developers for which output an automated diagnosis assistant would provide if the respondent designed the tool.
- Used open card sort to obtain categories
- Here, focus on categories hardly addressed by current tools
Debugging Tools Should…

- generate a **diagnosis or explanation why the error occurs** (25%)

- report the **most general environment or conditions** under which the bug can be reproduced (14%)

- visualize **divergence from the expected value** of a variable (10%)

- visualize **the range of expected values** for a given variable (4%)
Debugging Tools Should...

- highlight the symptoms and side-effects of an error (11%)
- classify the error according to its symptom in a category (14%)
- evaluate criticality of the symptoms (e.g., security risk) (2%)
Automated Repair

A Survey

• 18% of respondents would output an auto-generated patch as debugging aid.
How Do Developers Debug?

A Survey
An Experiment
A Benchmark
• Based on survey, we designed and conducted experiments with professional software developers to find out how they debug programs.
Experiment Goals

• How much **time** do developers spend on bug diagnosis and patching?

• What makes **difficult errors** so difficult?

• Is there a **single fault**, a single **diagnosis**, a single **patch**?

• How **correct** and plausible are the **fixes**?
Experiment Subjects

- Set up **Docker** virtual environment with most common development and debugging tools, including **gdb**, **vim**, and **Eclipse**

- Set up README file, 34 slides, and 10 tutorial videos

- Used **27 reproducible errors** in **find** and **grep** from COREBENCH (17k/19k LOC)
Demographics

- Participants with C experience from survey
- 1 researcher and 11 professional software engineers from six countries (Russia, India, Slovenia, Spain, Canada, and Ukraine)
- Paid 540 US$ each for time and effort
- Problems with German minimum wage law
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)

Debug this!
On average, participants spent 32 minutes diagnosing an error and 16 minutes patching it.
Single Diagnosis Assumption

- For each error, we asked participants to **provide a diagnosis**: the root cause of the error and the runtime actions leading to the error (with locations)

- **85%** of participants provide **essentially the same diagnosis** for an error.

<table>
<thead>
<tr>
<th>grep.5fa8c7c9</th>
<th>Error Type: Infinite Loop</th>
<th>Avg. Time: 38.8 min</th>
<th>Explanation: Moderately difficult</th>
<th>Patching: Slightly difficult</th>
<th>Correctness: 50%</th>
</tr>
</thead>
</table>

| grep.5fa8c7c9 | If grep is set to search for fixed strings (-F), the empty string is given (""), and the locale is UTF8, then grep runs indefinitely. 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:117-116) 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 return complete buffer until end of line (Regression because only match should be returned). |

- Is this what automated debugging tools should provide?
In their diagnosis of the error, participants on average reference 3–4 code regions.

- One suspicious statement does not suffice to understand the error.
- But one diagnosis could help!
Patch vs Fault Location

- Only 69% of submitted patches modify statements that are referenced in the bug diagnosis.
- Often, there are several ways to patch an error correctly, syntactically and semantically.
Correctness

• 97% (282/291) of the submitted patches pass the test case

• 58% (170/291) are actually correct
Bug Diagnosis Strategies

- (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.
Patch Effects

- 70% of patches affect **control flow**:
  - 63% change a branch condition
  - 19% modify loop or function flow
  - 43% add new branches

- 64% of patches affect **data flow**:
  - 30% change a variable
  - 39% add a statement;
    24% move one, 16% delete one
  - 2.8% introduce new functions
Implications

• **Program understanding** is crucial: Better documentation

• Events leading to failure involve **multiple steps**: Need automated event chains

• **Automated suggestions and patches** may not help with these problems
How Do Developers Debug?

A Survey

An Experiment

A Benchmark
A Debugging Benchmark

DBGBENCH contains 27 errors, each with
- failing test case
- simplified bug report
- the identified fault locations
- an explanation of the events leading to the error
- the time taken to understand and fix the error
- examples of correct and incorrect patches.
A Benchmark

grep.Sla8c/c9

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infinite Loop</td>
<td>The search function continues indefinitely, resulting in a runtime error.</td>
</tr>
<tr>
<td>Avg. Time: 38.8 min</td>
<td>The time taken to execute the program.</td>
</tr>
<tr>
<td>Patching: Slightly difficult</td>
<td>The difficulty of applying a fix to the program.</td>
</tr>
<tr>
<td>Correctness: 30%</td>
<td>The accuracy of the fix in terms of correctness.</td>
</tr>
</tbody>
</table>

Examples of Incorrect Fixes:
1. If grep is set to search for fixed strings (^-), the empty string is given (""), and the locale is UTF8, then grep runs indefinitely. When grep.execute 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 if 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 continued (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:**

   - Remove continue (Treating the Symptom).
   - Don’t reset beg (Regression because it breaks multibyte character handling).
   - Remove part of the check which causes is mb_middle to return true (Regression because it breaks multibyte character handling).
   - Do not compute match size but return complete buffer until end of line (Regression because only match should be returned).
A Debugging Benchmark

You can use the diagnoses in DBGBENCH to
• evaluate automated fault localization techniques
• evaluate automated bug diagnosis techniques
• evaluate automated repair techniques
You can use the data in DBGBENCH to
• measure how much faster developers can be if assisted with automated tools.
About DBG Bench

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.

Usage

DBGBENCH allows to evaluate novel automated debugging and patching techniques and assistants:

- Evaluating Fault Localization Techniques: 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.

- Evaluating Bug Diagnosis Techniques: 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.

- Evaluating Automated Repair Techniques: 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.

- Evaluating the Effectiveness of Debugging Assistants: 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.

Downloads

- Download the DBG Bench technical report titled: How Developers Diagnose and Repair Software Bugs

Contact:

SE chair at Saarland University
Marcel Böhme
Ezraeili O. Boramekun
Sudipta Chattopadhyay
Emanurho J. Ugheroghe
Andreas Zeiler

https://www.st.cs.uni-saarland.de/debugging/dbgbench/
Debugging Tools Should...

- generate a **diagnosis or explanation** why the error occurs (25%)
- report the **most general environment or conditions** under which the bug can be reproduced (14%)
- visualize **divergence from the expected value** of a variable (10%)
- visualize **the range of expected values** for a given variable (4%)

Implications

- **Program understanding** is crucial: Better documentation
- Events leading to failure involve **multiple steps**: Need automated event chains
- Automated suggestions and patches may not help with these problems

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