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

5
Author(s)
W. Eric Wong ; Ruizhi Gao ; Yihao Li ; Rui Abreu ; Franz Wotawa
View All Authors

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

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

second activity, *fault understanding*, involves understanding the root cause of the failure. Finally, *fault correction* is determining how to modify the code to remove such root cause. Fault localization, understanding, and correction are referred to collectively with the term *debugging*.

Debugging is often a frustrating and time-consuming experience that can be responsible for a significant part of the cost of software maintenance [25]. This is especially true for today's software, whose complexity, configurability, portability, and dynamism exacerbate debugging challenges. For this reason, the idea of reducing the costs of debugging tasks through techniques that can improve efficiency and effectiveness of such tasks is ever compelling. In fact, in the last few

KNOW YOUR CUSTOMER

New Approaches to Understanding Customer Value and Satisfaction

ROBERT B. WOODRUFF
AND
SARAH F. GARDIAL

How Do Developers Debug?

A Survey

An Experiment

A Benchmark

A Survey

A Survey

- Surveyed **developers** on
  - **time** spent on debugging
  - **familiarity** with debugged code
  - **debugging techniques** used
  - debugging techniques **needed**

A Survey

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.

Debugging Time

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?** *

Make sure it adds up to 100% :)
Mark only one oval per row.

	Less than 5%	5%	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%	More than 95%
Bug Reproduction													
Bug Diagnosis													
Bug Fixing													

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

Tool Support

11. **How often do you use the following Bug Diagnosis techniques?** *

Mark only one oval per row.

	Never	Rarely	Sometimes	Often	Always
Trace-based Debugging (using printing; e.g., println, log4c)					
Interactive or Online Debugging (using breakpoints; e.g., gdb, jdb)					
Post-Mortem or Offline Debugging (using core dumps and stack traces)					
Delta Debugging to minimize failure-inducing input (e.g., AskIgor)					
Regression Debugging to identify failure-inducing changes (e.g., git bisect)					
Statistical or Spectrum-based Debugging to find suspicious 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., Java DD)					

12. **Are there other *automated* Bug Diagnosis techniques not listed that you use Always or Often?**

Please specify in one to three words!

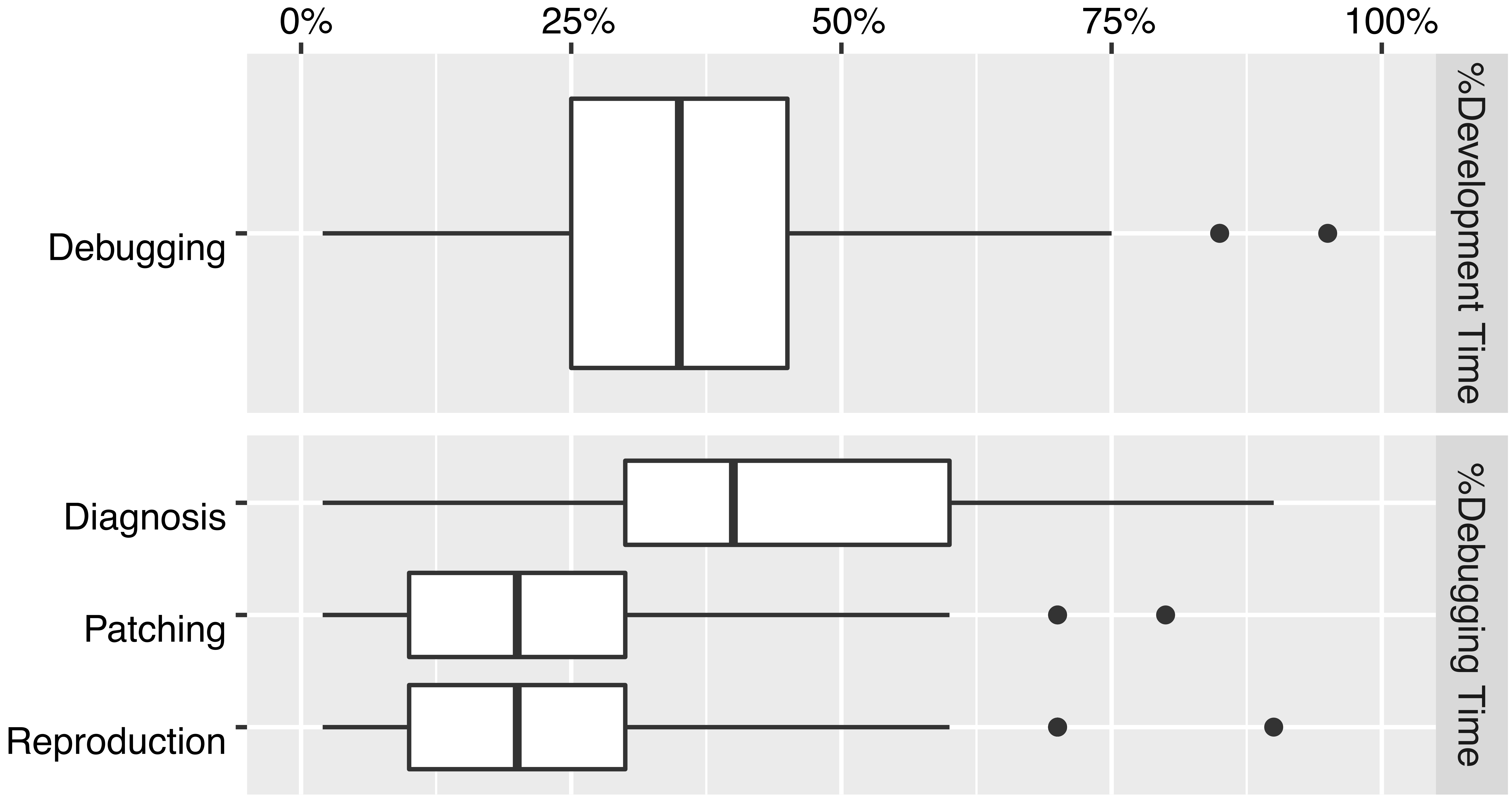
....

Demographics

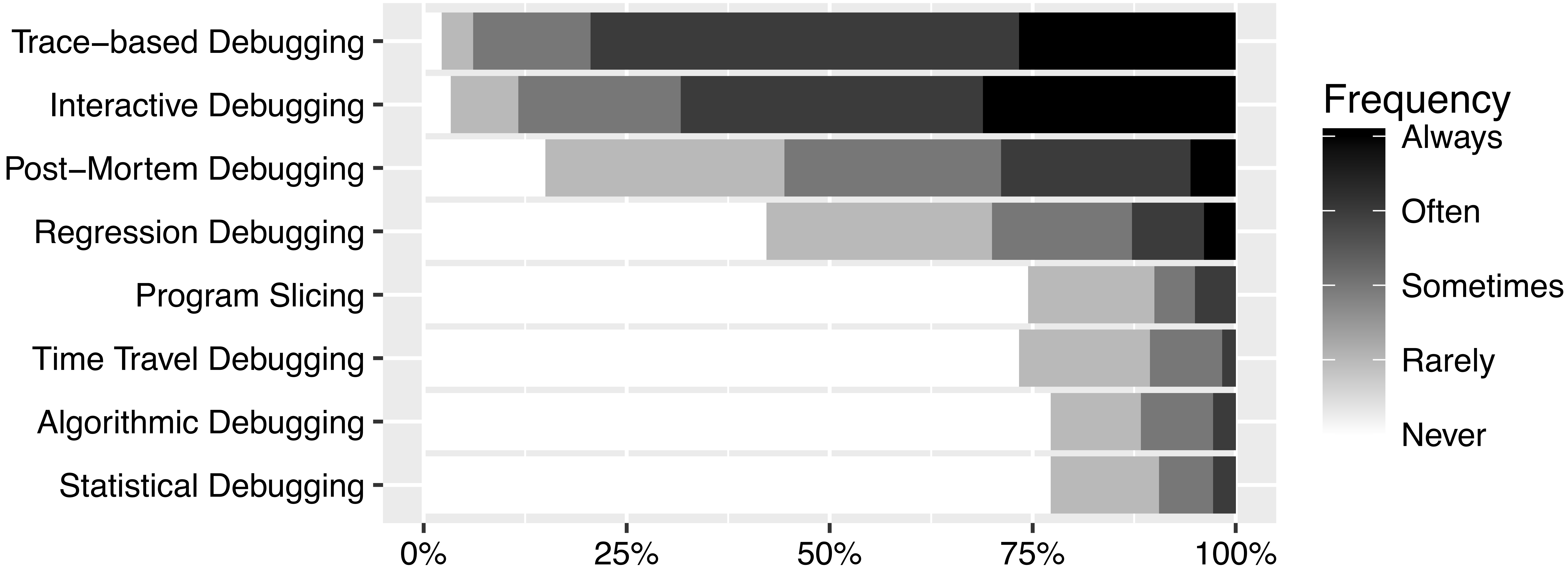
A Survey

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

Spending Time



Debugging Techniques Used



What do Developers need?

A Survey

- 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...

A Survey

- 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...

A Survey

- 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

An Experiment

An Experiment

- Based on survey, we **designed and conducted experiments with professional software developers** to find out how they debug programs.

Experiment Goals

An Experiment

- 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

An Experiment

- 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

An Experiment

- 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**

grep.5fa8c7c9 bug report

An Experiment

Hang in grep -F for empty string search

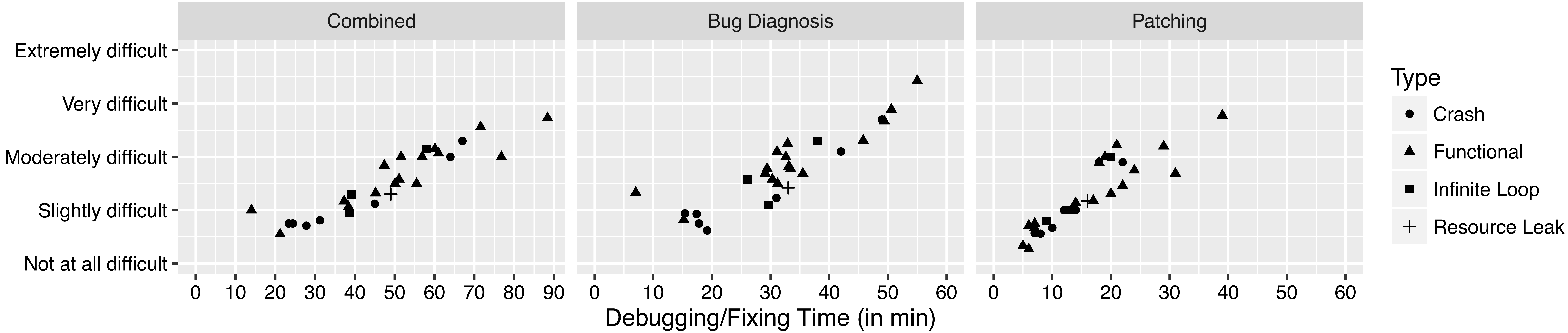
```
Searching with grep -F for an empty string in a
multibyte locals would freeze grep.

For example,
$ export LC_ALL=en_US.UTF-8
$ echo "abcd" | ./grep -F ""
(runs forever)
```

Debug this!

Time Spent

- 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.

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

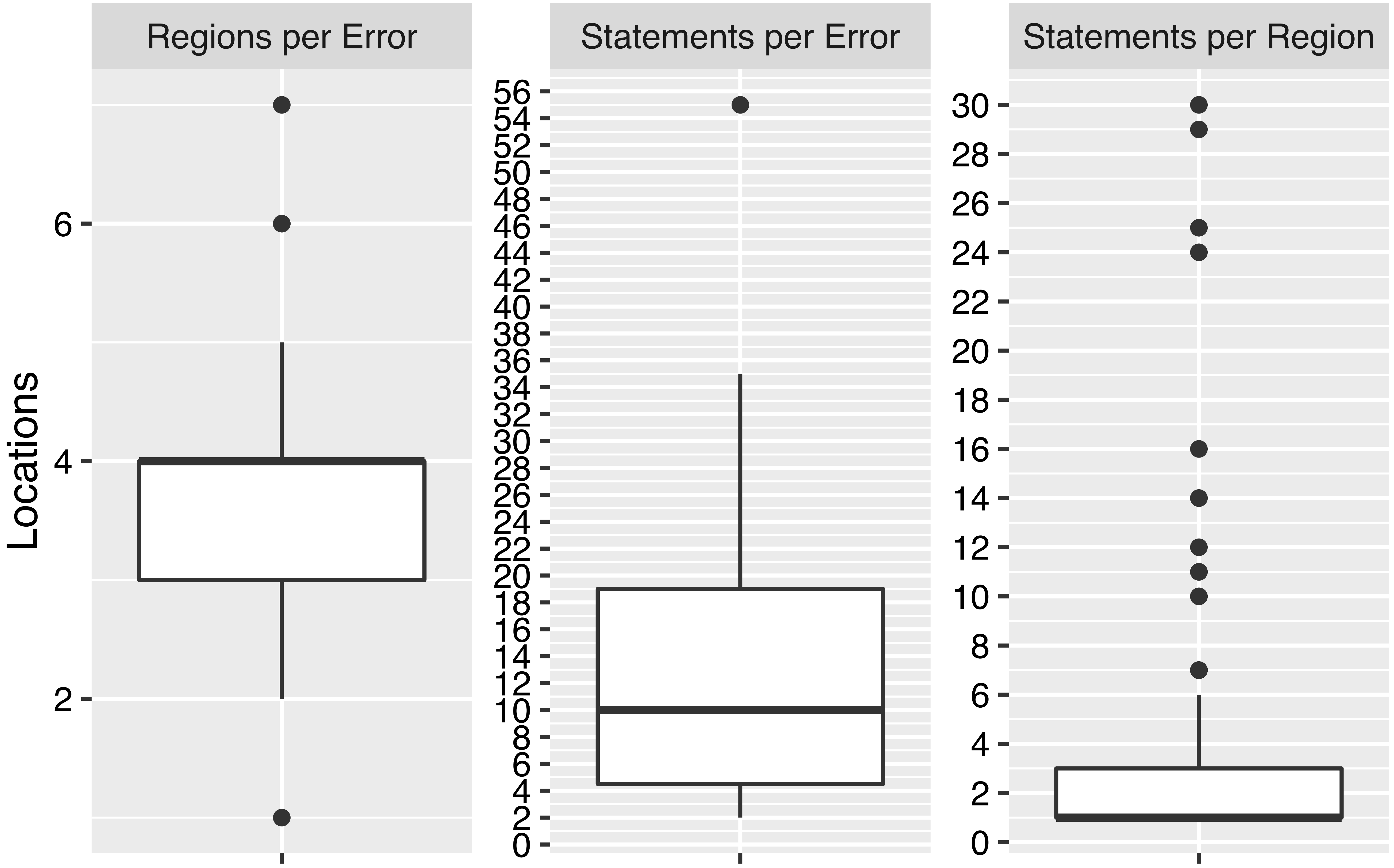
- Is this what automated debugging tools should provide?

Single Fault Assumption

- In their diagnosis of the error, participants on average **reference 3–4 code regions**

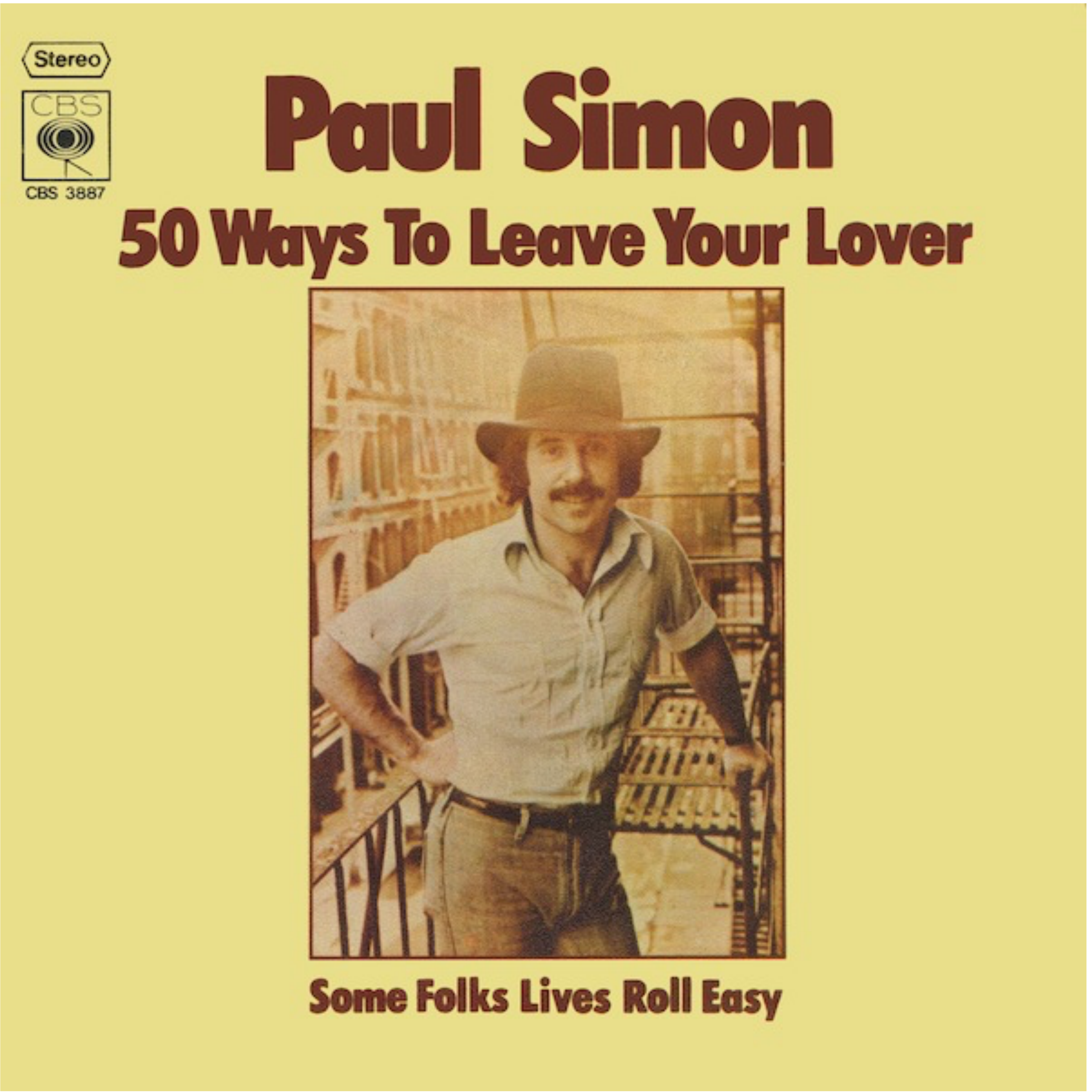
➡ **One suspicious statement does not suffice** to understand the error

➡ But one **diagnosis** could help!



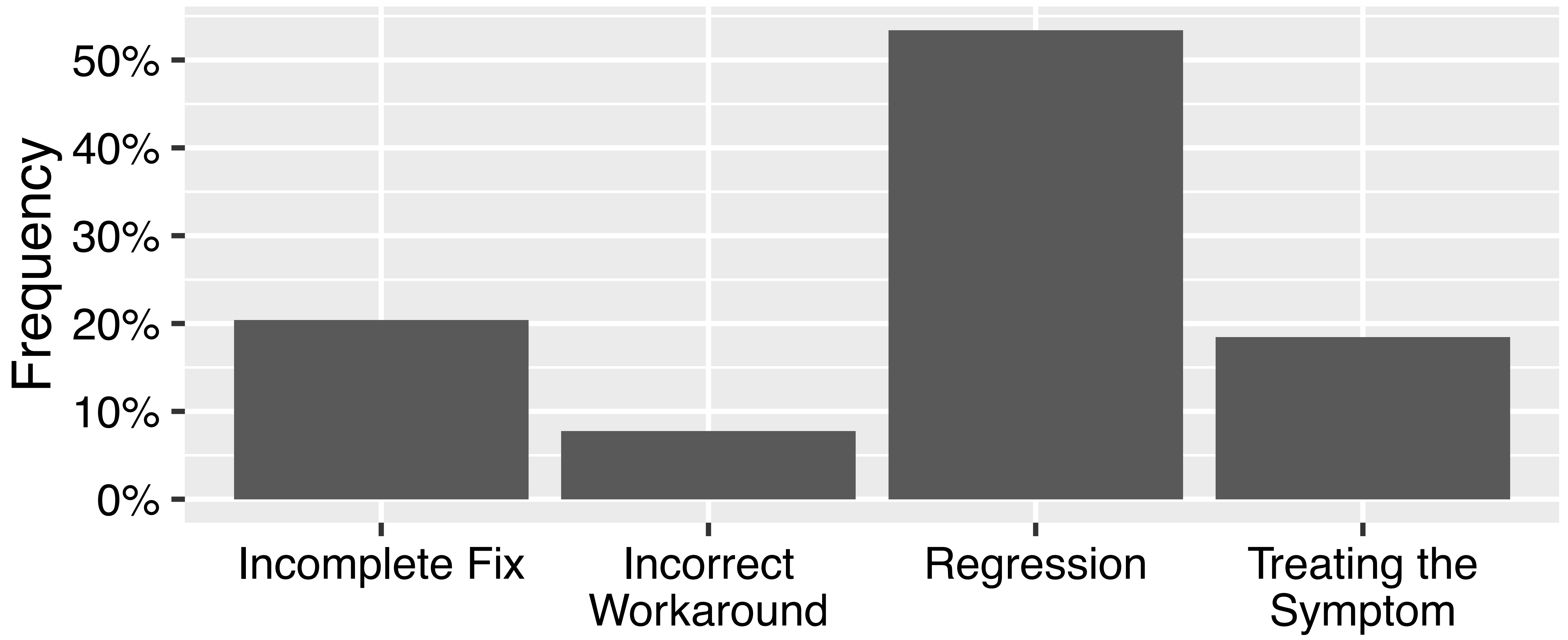
Patch vs Fault Location

- 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.



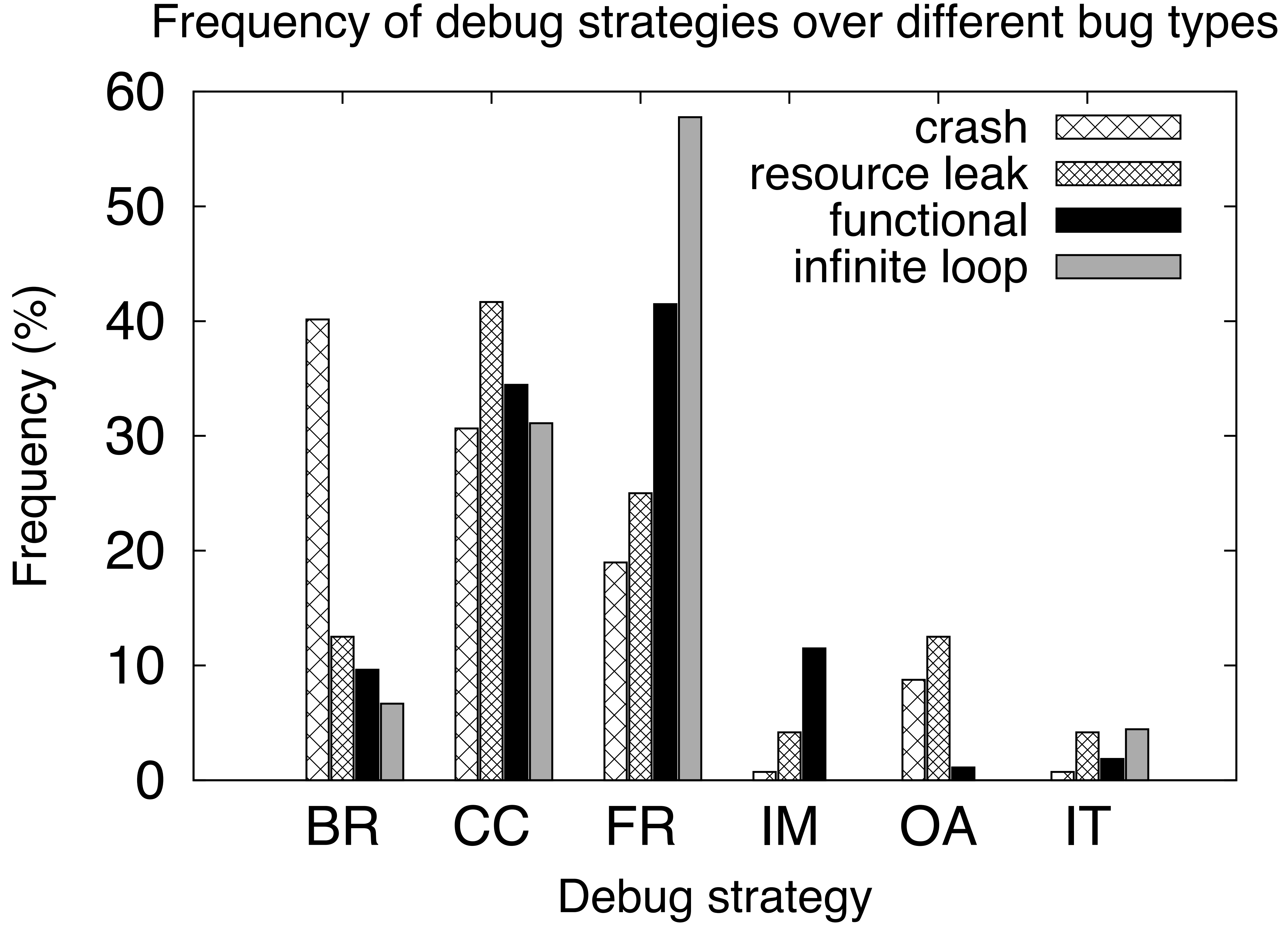
Correctness

- **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

An Experiment

- **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

A 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 Debugging Benchmark

A Benchmark

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).

A Debugging Benchmark

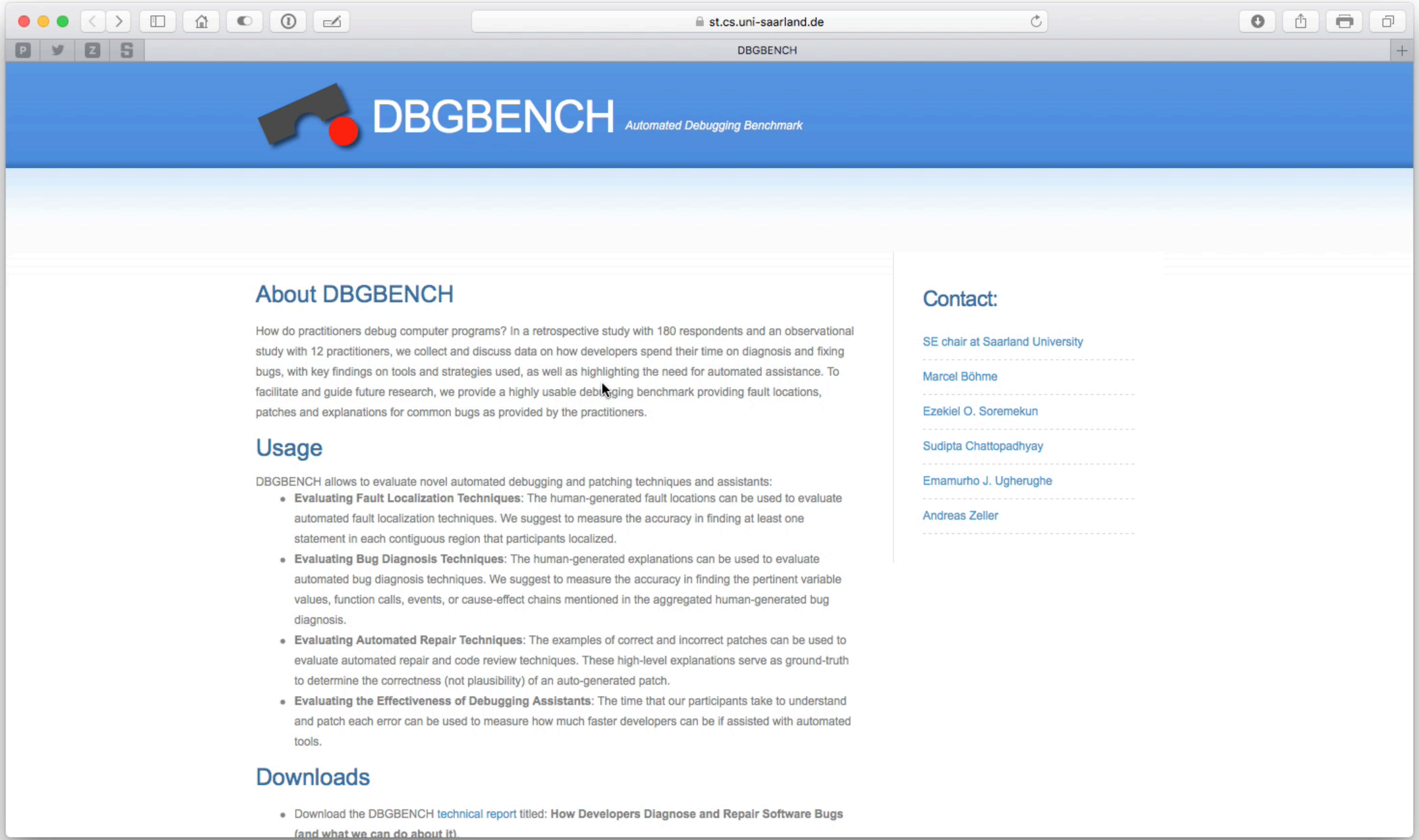
A Benchmark

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 DBGBENCH

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 DBGBENCH technical report titled: **How Developers Diagnose and Repair Software Bugs (and what we can do about it)**

Contact:

- SE chair at Saarland University
- Marcel Böhme
- Ezekiel O. Soremekun
- Sudipta Chattopadhyay
- Emamurho J. Ugherughe
- Andreas Zeller

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

Debugging Tools Should…

A Survey

- 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%)

grep.5fa8c7c9 bug report

An Experiment

```
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)
```

Implications

An Experiment

- **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

A Debugging Benchmark

A Benchmark

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).

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