Programmers Should Still Use Slices When Debugging

Ezekiel O. Soremekun
Software Engineering Chair
Saarland University
Saarbrücken, Germany
Email: soremekun@cs.uni-saarland.de

Marcel Böhme
School of Computing
National University of Singapore
Singapore
Email: marcel.boehme@acm.org

Andreas Zeller
Software Engineering Chair
Saarland University
Saarbrücken, Germany
Email: zeller@cispa.saarland

***Abstract*—What is the best technique for fault localization? In a study of 37 real bugs (and 37 injected faults) in more than a dozen open source C programs, we compare the effectiveness of statistical debugging against dynamic slicing—the first study ever to compare the techniques. On average, *dynamic slicing is more effective than statistical debugging,* requiring programmers to examine only 14% (42 lines) of the code before finding the defect, less than half the effort required by statistical debugging (30% or 157 lines). Best results are obtained by a *hybrid approach*: If programmers first examine the top five most suspicious locations from statistical debugging, and then switch to dynamic slices, they will need to examine only 11% (35 lines) of the code.**

I. Introduction

In the past 20 years, the field of *automated fault localization* has found considerable interest among researchers in Software Engineering. Given a program failure, the aim of fault localization is to suggest locations in the program code where a fault in the code causes the failure at hand. Locating a fault is an obvious prerequisite for removing and fixing it; and thus, *automated* fault localization brings the promise of supporting programmers during arduous debugging tasks. Fault localization is also an important prerequisite for *automated program repair,* where the identified fault locations serve as candidates for applying the synthesized patch [1]–[4].

The large majority of today's publications on automated fault localization fall into the category of *statistical debugging*, an approach pioneered 15 years ago by both Liblit et al. [5], [6] as well as Jones et al. [7]. A recent survey [8] lists more than 100 publications on statistical debugging in the past 15 years.

The core idea of statistical debugging is to take a set of passing and failing runs, and to record the program lines which are executed ("covered") in these runs. The stronger the correlation between the execution of a line and failure (say, because the line is executed only in failing runs, and never in passing runs), the more we consider the line as "suspicious".

As an example, let us have a look at the function `middle`, pioneered in [7] to introduce the technique. The `middle` function computes the middle of three numbers `x, y, z`; Figure 1 shows its source code as well as a few sample inputs. On most inputs, `middle` works as advertised; but when fed with `x = 2`, `y = 1`, and `z = 3`, it returns 1 rather than the middle value 2. Note that the statement in Line 8 is incorrect and should read `m = x`. Given the runs and the lines covered in each, statistical debugging assigns a suspiciousness score to each program statement. The suspiciousness of a

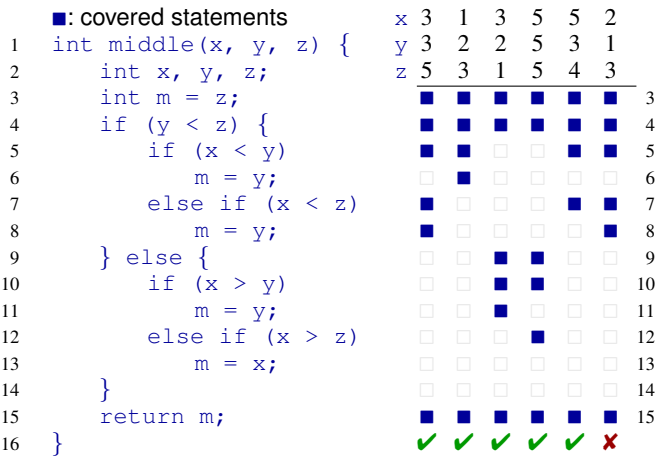

Fig. 1. Statistical debugging illustrated [9]: The `middle` function takes three values and returns that value which is greater than or equals the smallest and less than or equals the biggest value; however, on the input (2, 1, 3), it returns 1 rather than 2. Statistical debugging reports the faulty Line 8 as the most suspicious one, since the correlation of its execution with failure is the strongest.

statement is computed as a function on the number of times it is (not) executed by passing and failing test cases. The function itself differs for each statistical debugging technique. Since the statement in Line 8 is executed most often by the failing test case and least often by any passing test case, it is reported as most suspicious fault location.

Statistical debugging, however, is not the first technique to automate fault localization. In his seminal paper of 1985 "Programmers use slices when debugging" [10], Mark Weiser introduced the concept of a *program slice* composed of data and control dependencies in the program, and argued that during debugging, programmers would start from the location where the error is observed, and then proceed backwards along these dependencies to find the fault. In a debugging setting, the programmer would follow *dynamic* dependencies to find those lines that actually impact the location of interest in the *failing run*. In our example, she simply follows the dynamic dependency of Line 15 where the value of `m` is unexpected, and immediately reaches the faulty assignment in Line 8. Consequently, on the *example originally introduced to show the effectiveness of statistical debugging,* the older technique of dynamic slicing is just as effective. However, to the best of our knowledge, *no statistical debugging technique has ever been compared against any form of slicing.*

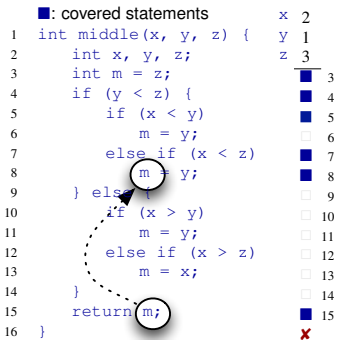

Fig. 2. Dynamic slicing illustrated [9]: The `middle` return value in Line 15 can stem from any of the assignments to `m`, but only those in Lines 3 and 8 are executed in the failing run. Following back the dynamic dependency immediately gets the programmer to Line 8, the faulty one.

In this paper, we take a localizable set of the COREBENCH suite of bugs—a set of 37 real, confirmed, and fixed bugs in a dozen programs, as well as 37 injected faults—and use them to compare both fault localization techniques against each other. Our takeaway findings are as follows:

1) **Top ranked locations in statistical debugging can pinpoint the fault.** If one is only interested in a *small set of candidate locations,* statistical debugging frequently outperforms dynamic slicing. In our experiments, looking at the top five locations only, statistical debugging would reveal 10% of all faults, whereas a dynamic backward slice from the output across five locations would detect only 5%. This result is important for *automatic repair techniques,* as the search for possible repairs can only consider a limited set of candidate locations; also, the search is not necessarily expected to succeed.
2) **If one *must* fix the bug, dynamic slicing is twice as effective.** In our experiments, locating faults along dynamic dependencies requires programmers to examine on average 14% of the code (42 LoC); whereas statistical debugging techniques require 30% (157 LoC). This is important for *human debuggers,* as they eventually must find and fix the fault: If they follow the dynamic slice from the failing output, they will find the fault much quicker than if they examine locations whose execution correlates with failure. On top, dynamic slicing needs only the failing run, whereas statistical debugging additionally requires multiple similar passing runs.
3) **Programmers can start with statistical debugging, but should quickly switch to dynamic slicing after a few locations.** In our experiments, it is a *hybrid* search strategy that works best: First consider the top locations of statistical debugging (if applicable), and then proceed along the dynamic slice. In our experiments, this hybrid strategy required programmers to examine only 11% of the code (35 LoC), improving over dynamic slicing alone.

The remainder of this paper is organized as follows. Section II and Section III introduce program slicing and statistical debugging, respectively. This is followed by a discussion of a hybrid strategy in Section IV where the developer switches to slicing after investigating the N most suspicious statements. Section V describes our experimental setting while the results are detailed in Section VI. Section VII closes with conclusion and consequences.

II. PROGRAM SLICING

It was more than three decades ago that Mark Weiser [10], [11] noticed that developers localize the root cause of a failure by following chains of statements starting from where the failure is observed. Starting from the symptomatic statement s where the error is observed, the developer would identify those program locations that directly influence the variable values or execution of s. This traversal continues until, transitively, the root cause of the failure (i.e., the *fault*) is found. This procedure allows her to investigate, in reverse, those parts of the program involved in the infected information-flow towards the location where the failure is first observed.

A. Static Slicing

Weiser developed *program slicing* as the first automated fault localization technique. A person marks the statement where the failure is observed (i.e., the failure's symptom) as *slicing criterion* C. To determine the potential impact of one statement onto another, the program slicer first computes the Program Dependence Graph (PDG) for the buggy program.

The *PDG* is a directed graph with nodes for each statement and an edge from a node s to a node s' if

1) statement s' is a conditional (e.g., an `if`-statement) and s is executed in a branch of s' (i.e., the values in s' *control* whether or not s is executed), or
2) statement s' defines a variable v that is used at s and s may be executed after s' without v being redefined at an intermediate location (i.e., the values in s' directly *influence the value* of the variables in s).

The first condition elicits *control dependence* while the second elicits *data dependence*. The PDG essentially captures the information-flow among all statements in the program. If there is no path from node n to node n', then the values of the variables at n' have definitely no impact on the execution of n' or its variable values.

The *static program slice* [11], [12] computed w.r.t. C consists of all statements that are reachable from C in the PDG. In other words, it contains all statements that potentially impact the execution and program states of the slicing criterion. Note that static slicing only removes those statements that are *definitely not* involved in observing the failure at C. The statements in the static slice may or may not be involved. Static program slices are often very large, containing an average of one third (33%) of the program's code [13].

B. Dynamic, Relevant, and Execution Slicing

A *dynamic program slice* [14], [15] is computed for a specific failing input t and is thus much smaller than a static slice. It is able to capture all statements that are *definitely* involved in computing the values that are observed at the location where the failure is observed for failing input t. Specifically,

the dynamic slice computed w.r.t. slicing criterion C for input t consists of all statements whose instances are reachable from C in the Dynamic Dependence Graph (DDG) for t. The *DDG* for t is computed similarly as the PDG only that the nodes are the statement *instances* in the execution trace $\pi(t)$. The DDG contains a separate node for each occurence of a statement in $\pi(t)$ with outgoing dependence edges to only those statement instances on which this statement instance depends in $\pi(t)$ [15]. However, an error is not explained only by the actual information-flow towards C. It is important also to investigate statements that could have contributed towards an alternative, potentially correct information-flow.

The *relevant slice* [16], [17] computed for a failing input t subsumes the dynamic slice for t but also captures the fact that the fault may be in *not* executing an alternative, correct path. It adds conditional statements (e.g., `if`-statements) that were executed by t and if evaluated differently may have contributed to a different value for the variables at C. It requires computing (static) potential dependencies. In the execution trace $\pi(t)$, a statement instance s *potentially depends* on conditional statement instance b if there exists a variable v used in s such that (i) v is not defined between b and s in trace $\pi(t)$, (ii) there exists a path σ from $\varphi(s)$ to $\varphi(b)$ in the PDG along which v is defined, where $\varphi(b)$ is the node in the PDG corresponding to the instance b, and (iii) evaluating b differently may cause this untraversed path σ to be exercised. Qi et al. [18] proved that the relevant slice w.r.t. C for t contains *all* statements required to explain the value of C for t.

The *approximate dynamic slice* [14], [19] is computed w.r.t. slicing criterion C for failing input t as the set of *executed* statements in the static slice w.r.t. C. The approximate dynamic slice subsumes the dynamic slice because there can be an edge from an instance s to an instance s' in the DDG for t only if there is an edge from statement $\varphi(s)$ to statement $\varphi(s')$ in the PDG. The approximate dynamic slice subsumes the relevant slice because it also accounts for potential dependencies: Suppose instance s potentially depends on instance b in execution trace $\pi(t)$. Then, by definition there exists a path σ from $\varphi(s)$ to $\varphi(b)$ in the PDG along at least one control- and one data-dependence edge (via the node defining v); and if $\varphi(s)$ is in the static slice, then $\varphi(b)$ is as well. Note that the approximate dynamic slice is

- *easier to compute* than dynamic slices (static analysis),
- *significantly smaller* than the static slice, and still
- *as "complete"* as the relevant slice.

In summary, *dynamic slice* $\subseteq$ *relevant slice* $\subseteq$ *approximate dynamic slice* $\subseteq$ *static slice*.

Figure 3 (a) and (b) show the static and the dynamic slice for the `middle` program, respectively. The slicing criterion was chosen as the return statement of the program—that statement where the failure is observed. As test case, we chose the single failing test case $\langle 2, 1, 3 \rangle$. In this example, the approximate dynamic slice matches exactly the dynamic slice.

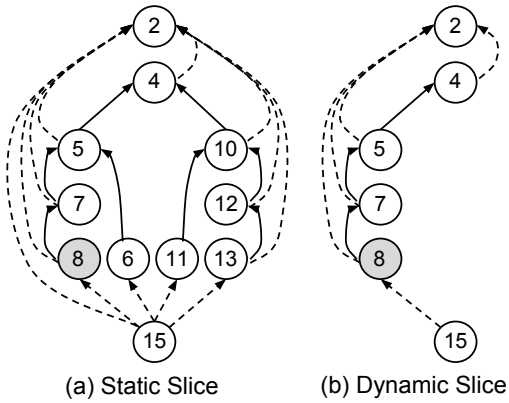

Fig. 3. Slicing Example: Nodes are statements in each line of the `middle` program (see Figure 1). Control-dependencies are shown as dashed lines while data dependencies are shown as concrete lines.

III. Statistical Debugging

It was one and a half decades ago that Jones et al. introduced the first statistical debugging technique, Tarantula [7], quickly followed by Liblit et al. [5], [6]. The main idea of *statistical debugging* is to associate the execution of a particular program element with the occurence of failure using so-called *suspiciousness measures*. Program elements (like statements, basic blocks, functions, components, etc.) that are observed more often in failed executions than in correct executions are deemed as more suspicious. A program element with a high suspiciousness score is more likely to be related to the root cause of the failure. Being able to suggest a fault location is not only important for human debuggers, but also for several *automated program repair techniques* [1]–[4], which first consider the highest ranked, most suspicious elements as patch location. Using a more effective debugging technique thus directly increases the effectiveness of such repair techniques.

The *effectiveness* of various statistical approaches to fault localization has been studied by several colleagues; for a recent survey on fault localization, see Wong et al. [8].

For our evaluation, we chose the three most popular statistical fault localization measures, Tarantula [7], [9], Ochiai [20], [21], and Jaccard [22]. They are either already used in recent automated repair techniques [2] or have been shown to improve automated repair techniques [23]. Other statistical fault localization techniques perform similarly well [23]–[25].

Figure 4 shows the scores computed for the executable lines in our motivating example. The statement in Line 8 is incorrect and should read `m = x;` instead. This statement is also the most suspicious according to all three statistical fault localization techniques. Notice that only twelve (12) lines are actually executable.

Evidently, in this example from Jones and Harrold [9], the faulty statement is also the most suspicious for all three statistical fault localization techniques. The scores for the faulty statement in Line 8 are $\mathit{tarantula}(s_8) = \frac{1}{1} / \left(\frac{1}{1} + \frac{1}{5}\right)$, $\mathit{ochiai}(s_8) = \frac{1}{\sqrt{1(1+1)}}$, and $\mathit{jaccard}(s_8) = \frac{1}{1+1}$.

```
1  int middle(x, y, z) {
2      int x, y, z;
3      int m = z;
4      if (y < z) {
5          if (x < y)
6              m = y;
7          else if (x < z)
8              m = y;
9      } else {
10         if (x > y)
11             m = y;
12         else if (x > z)
13             m = x;
14     }
15     return m;
16 }
```

Tarantula	Ochiai	Jaccard
0.500	0.408	0.167
0.500	0.408	0.167
0.500	0.408	0.167
0.625	0.500	0.250
0.000	0.000	0.000
0.714	0.578	0.333
0.833	0.707	0.500
0.000	0.000	0.000
0.000	0.000	0.000
0.000	0.000	0.000
0.000	0.000	0.000
0.000	0.000	0.000
0.000	0.000	0.000
0.500	0.408	0.167

Fig. 4. Statistical Fault Localization - Example

IV. A Hybrid Approach

We propose a new fault localization approach which leverages the strengths of both dynamic slicing and statistical debugging called the *hybrid approach*. The goal is to improve on the effectiveness of both approaches by harnessing the power of statistical correlation and dynamic program analysis. The hybrid approach first reports a few most suspicious statements before it reports the statements in the dynamic slice computed w.r.t. the symptomatic statement. This is inspired by the observation that programmers tend to transition to traditional debugging (i.e., finding those statements that impact the value of the symptomatic statement) after failing to locate the fault within the first N top-ranked most suspicious statements [26].

Specifically, the hybrid approach proceeds in two phases. In the first phase, it reports the N most suspicious statements, obtained from the ordinal ranking[1] of a statistical fault localization technique (e.g. Ochiai). Then, if the fault is not found, it proceeds to the second phase where it reports the symptom's dynamic backward dependencies. For CoreBench, we have empirically determined the best value of N to be five (5). Indeed, the hybrid approach with $N = 5$ has the programmer inspect the least number of statements before finding the fault.

We obtain the statistical fault localization ranking by executing the accompanying test suite of the program and tracking the coverage information for each statement and each test case. Furthermore, we obtain the dynamic backward dependencies via approximate dynamic slicing, by using the symptom of the failure as the slicing criterion. In the second phase, we account for statements that have already been reported in the first phase by not reporting them a second time.

A. Weakness of Statistical Debugging

The hybrid approach is capable of overcoming the weaknesses of statistical debugging. Statistical fault localization techniques are sensitive to the size and variance of the accompanying test suites [27]. Statistical debugging is less efficient when the accompanying test suite is small or achieves low coverage. To reduce the time wasted in search of the fault in these cases, the hybrid approach reports only the Top-N most suspicious statements, then proceeds to dynamic slicing.

[1]In ordinal ranking, lines with the same score are ranked by line number.

```
   ■: covered statements      x 3 2
1  int middle(x, y, z) {      y 3 1
2      int x, y, z;           z 5 3
3      int m = z;              ■   ■
4      if (y < z) {            ■   ■
5          if (x < y)          ■   ■
6              m = y;              
7          else if (x < z)     ■   ■
8              m = y;          ■   ■
9      } else {                    
10         if (x > y)              
11             m = y;              
12         else if (x > z)         
13             m = x;              
14     }                           
15     return m;               ■   ■
16 }                           ✔   ✘
```

Tarantula	Ochiai	Jaccard
0.500	0.707	0.333
0.500	0.707	0.333
0.500	0.707	0.333
0.000	0.000	0.000
0.500	0.707	0.333
0.500	0.707	0.333
0.000	0.000	0.000
0.000	0.000	0.000
0.000	0.000	0.000
0.000	0.000	0.000
0.000	0.000	0.000
0.500	0.707	0.333

Fig. 5. Test suite sensitivity of statistical debugging. Let us consider the `middle` function with a small test suite containing two test cases (3, 3, 5) and (2, 1, 3). Then, statistical debugging reports *all* executed lines 3, 4, 5, 7, 8, and 15 as the "most" suspicious statements, since they are all strongly correlated to the failure.

For instance, Figure 5 depicts for our motivating example how the effectiveness of statistical fault localization depends on the provided test suite. Given one passing and one failing test case, statistical debugging correlates all six executed statements for the failing test case—$(2, 3, 4, 7, 8, 15)$, as the *most* suspicious ranked statements. Conservatively, the programmer needs to inspect half the program statements before finding the fault. Although a large test suite and a high coverage is desirable for statistical fault localization, for real programs this is not always available. Often only one or two failing test cases are actually available [28].

Meanwhile, the hybrid approach with the same test suite improves the programmer's effectivenss. Assuming $N = 2$, the programmer inspects the first two statements before following the dependency from the symptomatic statement in Line 15. She finds the fault after inspecting only three statements. In contrast, using statistical debugging she would find the fault after investigating five statements (using ordinal ranking).

B. Weakness of Dynamic Slicing

Program slices can become very large [13]. Generally, programmers using dynamic slicing become ineffective when the fault is located relatively far away from the symptomatic statement. However, our proposed hybrid approach can overcome this limitation by leveraging statistical debugging which can point to any statement in the program however far from the symptomatic statement. This improves the chances of finding the fault quickly by first applying statistical correlation before dynamic analysis.

Figure 6 illustrates this weakness. This modified variant of the middle function contains another fault which is exposed by the same test suite. Since the return value for test case (1,2,3) is unexpected, we mark Line 15 as slicing criterion. In this example, the operator fault is located relatively far from the slicing criterion. The programmer following backward dependencies from the symptom has to inspect three statements (lines 8, 7, and 5) in addition to the slicing criterion. On the other hand, our hybrid approach with $N \geq 1$ has to inspect only a single statement before the fault is located.

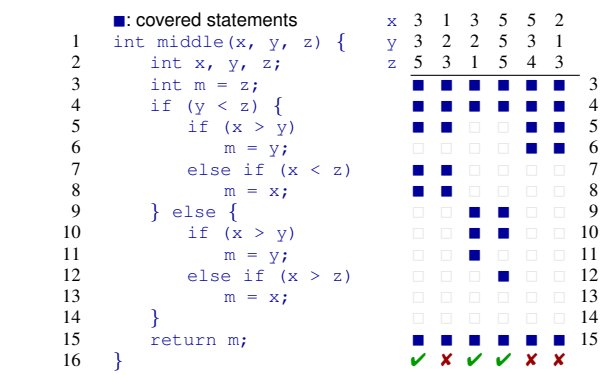

Fig. 6. Weakness of dynamic slicing. This is a variant of the faulty `middle` function with an operator fault in Line 5 (instead of Line 8). Following back the dynamic dependency gets the programmer to Line 5 (the fault) after inspecting 2–3 statements—(5,6) or (5,7,8) depending on the failing test case.

V. Experimental Setup

We evaluate the performance of statistical debugging and dynamic slicing in the framework of Steimann, Frenkel, and Abreu [29] where we fix the granularity of fault localization at *statement level* and the fault localization mode at *one-at-a-time*. In this practical setting with real errors and real test suites the provided test suites *may not be coverage adequate*. Fault localization effectiveness is evaluated as *relative wasted effort* based on the ranking of units in the order they are suggested to be examined.

A. Objects of Empirical Analysis

COREBENCH [28] is a collection of 70 real errors that were systematically extracted from the repositories and bug reports of four open-source software projects: Make, Grep, Findutils, and Coreutils.[2] These projects are well-tested, well-maintained, and widely-deployed open source programs for which the complete version history and all bug reports can be publicly accessed. All projects come with an extensive test suite. For each error, COREBENCH provides the patch that fixes the error and a test case that fails before but passes after the patch is applied.

Real Errors. Through a systematic analysis of 4×1000 recent code commits, Böhme and Roychoudhury [28] identified and validated 70 errors. 12% of the errors were fixed within a week while half stayed undetected and uncorrected for more than nine months up to 8.5 years. Eleven errors were fixed incorrectly. In these cases the error was indeed removed in the fixed version. Yet, up to three new errors were introduced that required further fixes. All errors were submitted unintentionally by experienced developers and package maintainers despite the large test suite and despite the long practice of code reviews where each patch is carefully checked by other developers before it is committed to the code repository. The repository, bug reports, test cases, and much more are publicly accessible and real in the sense that they were created in production rather than under scientific supervision.

[2]http://www.comp.nus.edu.sg/~release/corebench/

TABLE I
OBJECTS: OPEN SOURCE PROJECTS IN COREBENCH [28]

Project	Tools	Total Size	#Errors	#Excluded
`Coreutils`	98	83k LoC	22	7
`Findutils`	4	18k LoC	15	3
`Grep`	1	11k LoC	15	5
`Tcas`	41	60 LoC	41	4

Minimal Patches. The user-generated patches are used to identify those statements in the buggy version that are marked faulty. In fact, Renieris and Reiss [30] recommend to identify as faulty statements those that need to be changed to derive the (correct) program that does not contain the error. The authors of COREBENCH made sure that the provided patches are minimal such that they are not tangled with other, unrelated changes. For each error, we consider as buggy program that revision which exists right before the error's patch. Hence, only patched statements are considered faulty.

Slicing Criterion. All aspects of approximate dynamic slicing can be fully automated. To this end, as slicing criterion we chose the last statement that is executed or the return statement of the last function that is executed. For instance, when the program crashes because an array is accessed out of bounds, the location of the array access is chosen as slicing criterion. In our implementation the slicing criterion is automatically selected by a bash script running GDB.

Passing and Failing Test Cases. All projects come with a manually written test suite which checks corner cases and that previously fixed errors do not re-emerge. The complete test suite is executed with `make check` or `make tests`. Executing a single test case is more difficult. Each project comes with its own testing framework. While `find` uses `Dejagnu`,[3] `coreutils` and `grep` implement their own test framework in perl, where each executable file in the tests-folder can be considered a test case. For statistical debugging, we execute each of these (passing) test cases individually to collect coverage information. The failing test case is provided in COREBENCH.

Table I lists all objects and the studied errors. For our evaluation, we also used all injected errors for `Tcas` that is available from the Siemens programs[4] and *the* most well-studied subject according to a recent survey on fault localization [8]. From COREBENCH, we used all projects, except the `Make` project.[5]

In summary, for our automated evaluation we used 74 errors in dozens of programs from two well-known benchmarks. For TCAS, each artificially injected error has 1.7 faulty lines, 1571 passing and 37 failing test cases, on average. For COREBENCH, each real error has 25 faulty lines, 42 passing test cases and one failing test case, on average.

[3]www.gnu.org/software/dejagnu/

[4]http://www-static.cc.gatech.edu/aristotle/Tools/subjects

[5]`Frama-C`, our tool of choice to construct the Program Dependence Graph (PDG) cannot handle some recursive or variadic method calls that are abundant in Make. For this reason, we also excluded 2 errors in grep. Otherwise, we excluded an error if no coverage information could be generated (e.g., infinite loops; 9 errors) or the faulty statement could not be identified (e.g., patch only added statements; 4 errors). All other errors are used for our evaluation.

B. Measure of Localization Effectiveness

We measure *fault localization effectiveness* as the proportion of statements that do *not* need to be examined until finding the first fault. This allows us to assign a score of 0 for the worst performance (i.e., all statements must be examined) and 1 for the best. More specifically, we measure the *score* = $1 - p$ where p is the percentage of statements that needs to be examined before the first faulty statement is found. Not all failures are caused by a single faulty statement. In fact, only about 10% of failures are caused by a single statement while there is a long tail of failures that are substantially more complex [28]. Focusing on the first faulty statement found, the *score* measures the effort to find a good starting point to initiate the bug-fixing process rather than to provide the complete set of code that must be modified, deleted, or added to fix the failure. Wong et al. [8] motivates this measure of effectiveness and presents an overview of other measures.

Ranking. Both fault localization techniques produce a ranking. The developer starts examining the highest ranked statements and goes down the list until reaching the first faulty statement. To generate the ranking for *statistical debugging*, we list all statements in the order of their suspiciousness (as determined by the technique), most suspicious first. To generate the ranking for *approximate dynamic slicing*, given the statement c where the failure is observed, we rank first those statements in the slice that can be reached from c along one backward dependency edge. Then, we rank those statements that can be reached from c along two backward dependency edges, and so on. Generally, for both techniques, the $score$ is computed as

$$score = 1 - \frac{|S|}{|P|}$$

where S are all statements with the same rank or less as the highest ranked faulty statement and P is the set of all statements in the program. So, S represents the statements a developer needs to examine until finding the first faulty statement.

Multiple Statements, Same Rank. In most cases there are several statements that have the same rank as the faulty statement. We make the conservative assumption that a developer finds the faulty statement among other statements with the same rank *last*. More precisely, the ranking is a *modified competition ranking*, leaving gaps *before* the set of equal ranking items. Suppose, statements s_i where $1 \leq i \leq 4$ are ranked by index i except s_2 and s_3 are ranked equally. Then their ranks are $\langle 1, 3, 3, 4 \rangle$. This is in agreement with evaluations of fault localization techniques in previous work [8], [9], [24].

1) Dynamic Slicing Evaluation: We define the effectiveness of *approximate dynamic slicing*, the $score_{ads}$ according to Renieris and Reiss [30] as follows. Given a failing test case t, the symptomatic statement c, let ζ be the approximate dynamic slice computed w.r.t. c for t, let k_{min} be the minimal number of backward dependency edges between c and any faulty statement in ζ, and let $DS_*(c,t)$ be the set of statements in ζ that are reachable from c along at most k_{min} backward dependency edges. Then,

$$score_{ads} = 1 - \frac{|DS_*(c,t)|}{|P|}$$

Algorithmically, the $score_{ads}$ is computed by i) measuring the minimum distance k_{min} from the statement c where the failure is observed to any faulty statement along the backward dependency edges in the slice, ii) marking all statements in the slice that are at distance k_{min} or less from c, and iii) measuring the proportion of marked statements in the slice. This measures the part of code a developer investigates who follows backward dependencies of executed statements from the program location where the failure is observed towards the root cause of the failure.

For instance, the $score_{ads}$ for the approximate dynamic slice in our motivating example in Figure 3 is = $1 - \frac{1}{12} = 0.92$. The slicing criterion is $c = s_{15}$. The program size is $|P| = 12$. The faulty statement s_8 is ranked first. Statements s_7 and s_2 are both ranked third according to modified competition ranking. Statements s_5 and s_4 are ranked fourth and fifth, respectively, while the remaining, not executed (but executable) statements are ranked 12th.

2) Statistical Debugging Evaluation: We define the effectiveness of a *statistical fault localization* technique, the $score_{sfl}$ as follows. Given the modified competition ranking of program statements in P for test suite T according to their suspiciousness as determined by the statistical fault localization method, let r_f be the rank of the highest ranked faulty statement. Then,

$$score_{sfl} = 1 - \frac{r_f}{|P|}$$

Note that $score_{sfl}$ = 1 − *EXAM-score* where the well-known *EXAM-score* [9], [27] gives the proportion of statements that need to be examined until the first fault is found. Intuitively, the $score_{sfl}$ is its complement assigning 0 to the worst possible ranking where the developer needs to examine all statements before finding a faulty one.

For instance, $score_{sfl} = 1 - \frac{1}{12} = 0.92$ for our motivating example and all considered statistical debugging techniques. All statistical debugging techniques identify the faulty statement in Line 8 as most suspicious. So, there is only one top-ranked statement (Rank 1). But there are six statements with the lowest rank (Rank 12). If the fault was among one of these statements, the programmer might need to look at all statements of our small program `middle` before localizing the fault.

3) Hybrid Approach Evaluation: We define the effectiveness of the hybrid approach, the $score_{hyb}$ as follows. Let R be the set of faulty statements and H be the N most suspicious statements – sorted first by suspiciousness score and then by line numbers. Given the failing test case t and a statement c that is marked as symptomatic, we have

$$score_{hyb} = \begin{cases} \min(score_{sfl}, N) & \text{if } R \cap H \neq \emptyset \\ 1 - |H \cup DS_*(c,t)| \;/\; |P| & \text{otherwise} \end{cases}$$

Essentially, $score_{hyb}$ computes the score for the statistical fault localization technique if the faulty statement is within the first N most suspicious statements, and the score for approximate dynamic slicing while accounting for the statements already reported in the first phase. For instance, for $N = 2$ we have $score_{hyb} = 1 - \frac{1}{12} = 0.92$ for the motivating example in Figure 1 since the fault is amongst the N most suspicious statements. On the other hand, considering the same program with only two test cases (Figure 5), the hybrid approach has a score of 0.75 since the faulty statement is *not* amongst the N most suspicious statements. The score for the hybrid technique is better than that of statistical debugging (0.58).

C. Implementation and Infrastructure

We performed the experiments on the COREBENCH virtual machine running Ubuntu Linux 14.04. The VM was running on a MacBook Pro (Retina, 15-inch, Mid 2015) with a 2.5GHz Intel Core i7 CPU and 16GB of main memory.

1) Statistical Debugging Implementation: The statistical debugging tool was implemented using two bash scripts with several standard command line tools, notably `gcov`,[6] `git-diff`,[7] `gdb`,[8] and `bc`.[9] COREBENCH allows to implement a so-called `analyze`-script which is executed for each error on the version directly before the error was fixed. The differencing tool `git-diff` identifies those lines in the buggy program that were changed in the patch. If the patch only added statements, we cannot determine a corresponding faulty line. Seven errors were thus excluded from the evaluation.[10] The code coverage tool, `gcov` identifies those lines in the buggy program that are covered by an executed test case. When the program crashes, `gcov` does not emit any coverage information. If the crash is *not* caused by an infinite loop, it is sufficient to run the program under test in `gdb` and force-call the `gcov`-function from `gdb` to write the coverage information once the segmentation fault is triggered. This was automated as well. However, for eight cases, no coverage information could be generated due to infinite recursion.[11] `Gcov` also gives the number of *executable* statements in the buggy program (i.e., $|P|$). Finally, `bc` is an arbitrary precision numeric processing language and was used to compute the fault localization effectiveness.

2) Dynamic Slicing Implementation: The approximate dynamic slice is computed using `Frama-C`,[12] `gcov`, `git-diff`, `gdb`, and several python libraries. Given the preprocessed source files of a C program, `Frama-C` computes the static slices for each function and their call graphs as DOT files. The `gcov`-tool determines the executed/covered statements in the program. The `git-diff`-tool determines the changed statements in the patch and thus the faulty statements in the program. The `gdb`-tool allows to derive coverage information even for crashing inputs and to determine the slicing criterion as the last executed statement. Our python script intersects the statements in the static slice and the set of executed statements to derive the approximate dynamic slice. We use the python libraries `pygraphviz`[13], `networkxx`,[14] and `matplotlib`[15] to process the DOT files and compute the *score* for the approximate dynamic slice.

3) Hybrid Approach Implementation: The hybrid approach is implemented simply as a combination of both tools. If the top-N most suspicious statements do not contain the fault, the dynamic slicing component is informed about the set of statements already inspected in the first phase.

VI. RESULTS

A. Research Questions

In this study, we seek to answer these research questions:

- **RQ.1** Given a real error, which technique is more likely to be more effective at fault localization (*odds ratio*)? Which technique allows to investigate the least number of statements before finding the fault, on average (*mean*)?
- **RQ.2** *Cumulative Frequency*. Given a real error and assuming that a programmer is willing to investigate at most N statements, which technique performs best?
- **RQ.3** *Sensitivity*. How many most suspicious statements should the programmer inspect before switching to slicing?
- **RQ.4** Is there a difference between evaluating a fault localization technique on artificial versus real errors?

B. Presentation

For the presentation and evaluation of our results, we use three measures and metrics. *Odds Ratio* "is a measure of how many times greater the odds are that a member of a certain population will fall into a certain category than the odds are that a member of another population will fall into that category" [31]. We use odds ratio to assess whether fault localization technique A is more effective than fault localization technique B: Let b the number of successes for B, and $n = a + b$ the total number of successes. Then, the odds ratio ψ is calculated as

$$\psi = \left(\frac{a+\rho}{n+\rho-a}\right) \Big/ \left(\frac{b+\rho}{n+\rho-b}\right)$$

where ρ is an arbitrary positive constant (e.g., $\rho = 0.5$) used to avoid problems with zero successes. There is no difference between the two algorithms when $\psi = 1$ while $\psi > 1$ indicates that technique A has higher chances of success. For example, an odds ratio of 5 means that fault localization technique A is five times more likely to be successful (i.e., more effective as compared to B) at fault localization than B.

[6] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[7] https://git-scm.com/docs/git-diff
[8] https://www.gnu.org/software/gdb/documentation/
[9] https://www.gnu.org/software/bc/manual/html_mono/bc.html
[10] `core.2e636af1`, `core.a6a447fc`, `core.64d4a280`, `find.24bf33c0`, `tcas.13`, `tcas.14`, `tcas.36`, `tcas.38`.
[11] `core.51a8f707`, `core.61de57cd`, `core.8f976798`, `core.d461bfd2`, `find.ff248a20`, `grep.3220317a`, `grep.5fa8c7c9` and `grep.db9d6340`.
[12] http://frama-c.com/
[13] https://pygraphviz.github.io/
[14] https://networkx.github.io/
[15] http://matplotlib.org/

TABLE II

FAULT LOCALIZATION EFFECTIVENESS FOR APPROXIMATE DYNAMIC SLICING, STATISTICAL DEBUGGING, AND THE HYBRID APPROACH. FOR EACH PROJECT, WE SHOW THE MEAN EFFECTIVENESS μ, THE ODDS RATIO ψ (STATISTICALLY SIGNIFICANT VALUES IN BOLD), AND THE MANN-WHITNEY U TEST. WE COMPARE (1) SLICING VS. STATISTICAL DEBUGGING, (2) SLICING VS. HYBRID, AND (3) HYBRID VS. STATISTICAL DEBUGGING.

Object	Hybrid	Slicing	Statistical Debugging			Odds Ratio ψ			Mann-Whitney U		
	N = 5	μ	TARANTULA	OCHIAI	JACCARD	①	②	③	①	②	③
COREBENCH	0.89	0.86	0.70	0.70	0.70	**2.62**	0.77	**3.00**	< 0.05	0.23	< 0.05
`core`	0.83	0.82	0.73	0.73	0.73	1.46	1.13	1.46	0.32	0.68	0.18
`find`	0.94	0.92	0.71	0.71	0.71	**1.89**	0.53	**2.71**	< 0.05	0.45	< 0.05
`grep`	0.91	0.88	0.63	0.63	0.63	**21.00**	0.69	**21.00**	< 0.05	0.54	< 0.05
`tcas`	0.68	0.61	0.50	0.66	0.59	0.62	1.81	0.85	0.12	0.44	0.41

Besides odds ratio, we use the well known *Mann-Whitney* U measure to show whether any performance difference between both techniques is statistically significant. Finally, we use *cumulative frequency curves,* a running total of frequencies, to show the percentage of errors that require examining up to a certain percentage of program locations. The percentage locations examined is plotted on a log-scale because the difference between examining 5 to 10 locations is more important than difference between examining 1005 to 1010 locations.

RQ.1 Fault Localization Effectiveness

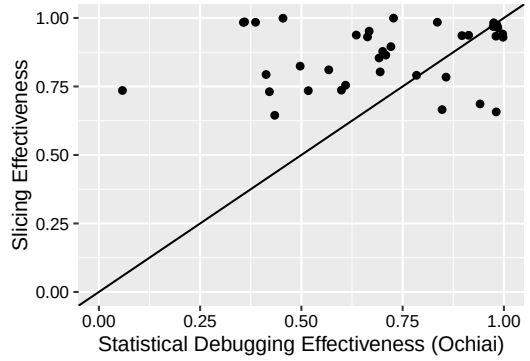

Fig. 7. Direct comparison of fault localization effectiveness between statistical debugging (Ochiai) versus approximate dynamic slicing for COREBENCH.

Odds Ratio. *For all real errors in our study, a programmer is about three (3) times more likely to find the fault location early if she starts with approximate dynamic slicing instead of statistical debugging but 28% less likely when compared to the hybrid approach* (N = 5). In other words, the majority of real bugs is best localized by the hybrid approach. We let Ochiai represent statistical debugging when computing odds ratio and Mann-Whitney test since its effectiveness is indeed representative of the other statistical techniques. As we can see in Table II, the odds ratio for all projects in COREBENCH is strictly in favor of the hybrid approach and approximate dynamic slicing ($\psi > 1$). The high odds ratio for `grep` is explained by slicing and the hybrid approach being more effective than statistical debugging in all ten cases. For `find` slicing is more effective than statistical debugging in eight out of twelve cases. For `core` slicing is more effective in nine out of fifteen cases.

Figure 7 shows a direct comparison of the scores computed for slicing and statistical debugging. The scatter plot shows for each error the effectiveness score of statistical debugging on the x-axis and the effectiveness score of slicing on the y-axis. Errors plotted above the line are better localized using slicing. We can see that slicing performs consistently above 60% and that most data points lie above the line at a significant distance.

Mean. *For all real errors in our study, a programmer using approximate dynamic slicing needs to examine less than half (47%) of those statements that she would need to examine if she used statistical debugging.*[16] If she used the hybrid approach she would need to examine only a third of the statements she would need to examine if she used statistical debugging. Table II shows that slicing is about 16 percentage points more effective than statistical debugging for the real errors in COREBENCH, on average.

RQ.2 Cumulative Frequency

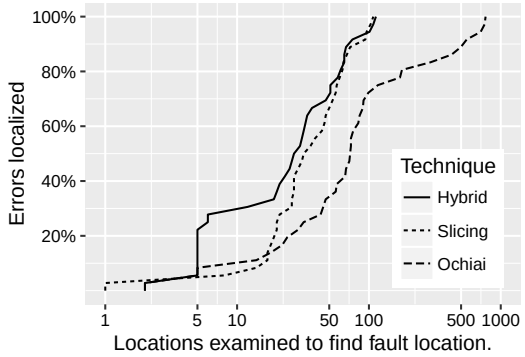

Fig. 8. Cumulative frequency of locations to be examined for each error for statistical debugging (Ochiai), approximate dynamic slicing, and the hybrid approach (N = 5) using the real errors in COREBENCH.

If the programmer is willing to inspect no more than 5 to 15 statements, statistical debugging performs better than approximate dynamic slicing. Otherwise, approximate dynamic slicing performs better. Suppose, a programmer is willing to inspect no more than the Top-10 most suspicious statements. She will be able to localize the fault for 10% of all errors if she used statistical debugging, compared to 5% for approximate dynamic slicing. If, however, the programmer is

[16]Percentage improvement is measured as $\frac{1-0.86}{1-0.70}$. Note that *score* by itself gives the number of statements that need *not* be examined.

patient enough to inspect 50 statements and used approximate dynamic slicing, she would localize the fault for 70% of the errors, compared to 30% if she used statistical debugging. The hybrid approach performs best independent of how many statements a programmer is willing to inspect.

RQ.3 Sensitivity of Hybrid Approach w.r.t. N

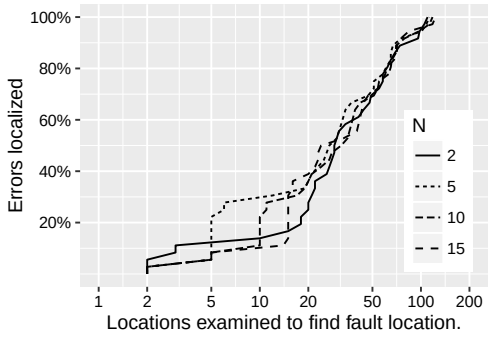

Fig. 9. Cumulative frequency of locations to be examined for each real error using the hybrid approach when switching from statistical debugging to slicing after examining the $N \in \{2, 5, 10, 15\}$ most suspicious statements.

When the programmer switches to slicing after investigating the $N = 5$ most suspicious statements, she can usually localize more errors than if she switched after investigating less or more suspicious statements (cf. Figure 9). Note that the hybrid approach degenerates to approximate dynamic slicing when $N = 0$ and to statistical debugging when N is large (e.g., program size). So, a good value of N lies somewhere in between those extremes. For the errors in COREBENCH, we determine $N = 5$ to be a good value.

RQ.4 Real vs. Artificial Errors

A recent survey of the statistical fault localization literature [8] identifies the Siemens benchmark of artificially injected errors and specifically Tcas as subjects that were *used most often for the evaluation* of fault localization effectiveness. The Siemens benchmarks come with very large test suites and a number of variants of small programs containing artificially injected errors. The test suites contain large numbers of failing test cases. Our preliminary results suggest that this poses a threat to the generality of the conclusions.

Observation 1. While for the artificial errors in Tcas there appears to be a significant difference in the performance of each statistical debugging technique, for the real errors in COREBENCH *all three techniques perform exactly the same* (see Figure 10). We would explain this observation with the abundance of failing test cases for Tcas. For COREBENCH, there is always only one or two failing test cases that can be used to correlate test case failure to the suspiciousness of a program statement. In fact, Abreu et al. [27] investigate the sensitivity of the effectiveness of statistical techniques on the provided passing and failing test cases and establish that the suspiciousness scores stabilize starting from an average six (6) failing and twenty (20) passing test cases. In a realistic setting so many failing test cases may not be available.

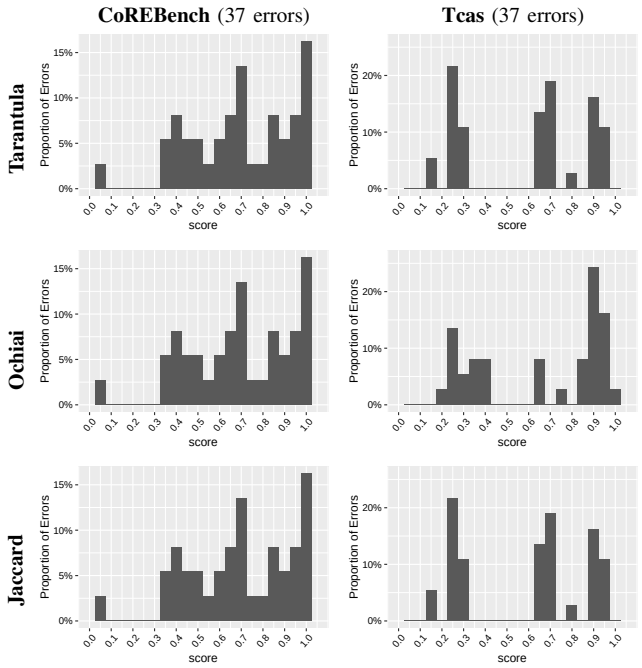

Fig. 10. Histograms of scores of three statistical fault localization techniques evaluated on the real errors in COREBENCH and the injected errors in TCas.

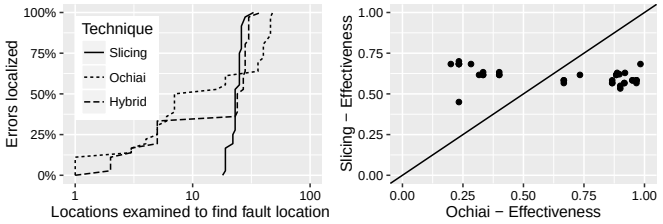

Fig. 11. Cumulative frequency curves for all techniques (left) and comparison of the effectiveness of slicing and statistical debugging (right) for Tcas.

Observation 2. For Tcas, neither statistical debugging nor approximate dynamic slicing show a significant advantage. The odds ratio in Table II shows that it is less likely that slicing outperforms statistical debugging than it is to fare the same or worse than statistical debugging. This is also evident in Figure 11. In the chart on the right, we can see that approximate dynamic slicing consistently achieves a score between 0.5 and 0.75 while the scores for statistical debugging varies significantly. We also note that the Mann-Whitney U test fails, suggesting no statistically significant difference between the effectiveness of approximate dynamic slicing versus statistical debugging for Tcas.

The Tcas faults demonstrate the circumstances under which statistical debugging can be better than dynamic slicing, namely if there is a large distance between fault and its manifestation—and, consequently, the program slice. However, we are not aware that Tcas or the manually injected faults were chosen and designed to be representative in that regard; we would expect the real faults from COREBENCH to much better fill that role.

C. Threats to Validity

We discuss the threats to validity for this fault localization study within the framework of Steimann et al. [29].

External validity refers to the extent to which the reported results can be generalized to other objects which are not included in the study. The most immediate threats to external validity are the following: *EV.1) Heterogeneity of Probands*. The quality of the test suites provided by the object of analysis may vary greatly which hampers the assessment of accuracy for practical purposes. However, in our study the test suites are well-stocked and maintained. All projects are GNU open source C programs which are subject to common measures of quality control, such as code review and providing a test case with bug fixes and feature additions. *EV.2) Faulty Versions and Fault Injection*. For studies involving artificially injected faults, it is important to control the type and number of injected faults. Test cases become subject to accidental fault injection. Some failures may be spurious. However, in our study we use real errors that were introduced (unintentionally) by real developers. Failing test cases are guaranteed to fail because of the error. *EV.3) Language Idiosyncrasies*. Indeed, our objects are well-maintained open-source C projects containing real errors typical for such projects. However, for instance errors in projects written in other languages, like Java, or in commercially developed software may be of different kind and complexity. Hence, we cannot claim generality for all languages and suggest to reproduce our experiments for real errors in projects written in other languages as well.

Construct validity refers to the degree to which a test measures what it claims, or purports, to be measuring. The most immediate threats to construct validity are the following: *CV.1) Measure of Effectiveness*. Conforming to the standard [8], we measure fault localization effectiveness as ranking-based relative wasted effort. The technique that ranks the faulty statement higher is considered more effective. Parnin and Orso find that "programmers will stop inspecting statements, and transition to traditional debugging, if they do not get promising results within the first few statements they inspect" [26]. However, Steimann et al. [29] insist that one may question the *usefulness* of fault locators, but measures of ranking-based relative wasted effort are certainly necessary for evaluating their performance, particularly in the absence of the subjective user as the evaluator. *CV.2) Implementation Flaws*. Tools that we used for the evaluation process may be inaccurate. Despite all care taken, our implementation of the three studied statistical fault localization techniques, or of approximate dynamic slicing, or of the empirical evaluation may be flawed or subject to random factors. However, we make all scripts and results available online for public scrutiny.

VII. Conclusion and Consequences

As it comes to debugging, *dynamic slices remain the technique of choice for programmers.* Suspicious statements, as produced by statistical debugging, can provide good starting points for an investigation; but beyond the top-ranked statements, following dependencies is much more likely to be effective. As it comes to teaching debugging, as well as for interactive debugging tools, we therefore postulate that following dependencies should remain the primary method of fault localization—it is a safe and robust technique that will get you towards the goal. Also keep in mind that our treatment of dynamic slicing is still conservative—we use approximate dynamic slicing, and not real dynamic slicing, and our strategy starts at the output, proceeding backwards, rather than following an algorithmic approach [32] where programmers would check intermediate results for correctness.

For automated repair techniques, the picture is different. Since current approaches benefit from a small set of suspicious locations, focusing on a small set of top ranked locations, as produced by statistical debugging, remains the strategy of choice. Still, automated repair tools could benefit from static and dynamic dependences just as human debuggers.

We found it surprising that of the more than 100 publications on statistical debugging, none compare against its older sibling, dynamic slicing. It is true that among all fault localization techniques, statistical debugging is one of the least demanding. However, dynamic slicing demands even less, in particular in the approximate form used in this paper. Besides static analysis, which as a compiler prerequisite can be assumed as a given, it does its job with just the coverage information of the failing run. As the tools and techniques we use in our comparison have been available since the first papers on statistical debugging were published, a comparison as conducted in this paper could and should have been called for by reviewers a long time ago.

While easy to deploy, the techniques discussed in this paper should by no means be considered the best of fault localization techniques. *Experimental techniques* which reduce inputs [33], [34] or executions [35] may dramatically improve fault localization by focusing on relevant parts of the execution. *Symbolic techniques* also show a great potential—such as the technique of Jose and Majumdar, which "quickly and precisely isolates a few lines of code whose change eliminates the error" [36]. The key challenge of automated fault localization will be to bring the best of the available techniques together in ways that are *applicable* to a wide range of programs and *useful* for real programmers, who must fix their bugs by the end of the day.

Additional material. All of our scripts, tools, benchmarks, and results are freely available to support scrutiny, evaluation, reproduction, and extension at the project site:

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

Acknowledgments

We are extremely grateful to the several colleagues who have carefully reviewed and commented on our scripts, results, and consequences. Thanks to you all!

This work was funded by Deutsche Forschungsgemeinschaft, Project "Extracting and Mining of Probabilistic Event Structures from Software Systems (EMPRESS)".

REFERENCES

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

[2] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 772–781.

[3] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 802–811.

[4] 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.

[5] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken, "Statistical debugging of sampled programs," in *Advances in Neural Information Processing Systems*, 2003, pp. 9–11.

[6] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, 2005, pp. 15–26.

[7] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02, 2002, pp. 467–477.

[8] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, p. preprint, 2016.

[9] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05, 2005, pp. 273–282.

[10] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, Jul. 1982.

[11] ——, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81, 1981, pp. 439–449.

[12] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, no. 3, pp. 121–189, 1995.

[13] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 2, Apr. 2007.

[14] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, no. 3, pp. 155–163, Oct. 1988.

[15] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI '90, 1990, pp. 246–256.

[16] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental regression testing," in *Proceedings of the Conference on Software Maintenance*, ser. ICSM '93, 1993, pp. 348–357.

[17] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7, 1999, pp. 303–321.

[18] D. Qi, H. D. T. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 32:1–32:41, Oct. 2013.

[19] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Efficient debugging with slicing and backtracking," Purdue University, Tech. Rep., 1990.

[20] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC'06, 2006, pp. 39–46.

[21] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, ser. TAICPART-MUTATION '07, 2007, pp. 89–98.

[22] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, ser. DSN '02, 2002, pp. 595–604.

[23] F. Y. Assiri and J. M. Bieman, "Fault localization for automated program repair: effectiveness, performance, repair correctness," *Software Quality Journal*, pp. 1–29, 2016.

[24] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10, 2010, pp. 1–10.

[25] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist," Department of Computer Science, University College London, Tech. Rep., 2014.

[26] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 199–209. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001445

[27] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009.

[28] M. Böhme and A. Roychoudhury, "Corebench: Studying complexity of regression errors," in *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA, 2014, pp. 105–115.

[29] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013, 2013, pp. 314–324.

[30] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. IEEE, 2003, pp. 30–39.

[31] R. Grissom and J. Kim, *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum, 2005.

[32] E. Y. Shapiro, *Algorithmic Program DeBugging*. Cambridge, MA, USA: MIT Press, 1983.

[33] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/32.988498

[34] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel, "On the use of delta debugging to reduce recordings and facilitate debugging of web applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 333–344. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786846

[35] M. Burger and A. Zeller, "Minimizing reproduction of software failures," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011, pp. 221–231.

[36] M. Jose and R. Majumdar, "Cause clue clauses: Error localization using maximum satisfiability," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 437–446. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993550