Empirical Software Engineering (2021) 26: 51
https://doi.org/10.1007/s10664-020-09931-7

Locating faults with program slicing: an empirical analysis

Ezekiel Soremekun[1] · **Lukas Kirschner**[2] · **Marcel Böhme**[3] · **Andreas Zeller**[2]

Accepted: 23 December 2020 / Published online: 1 April 2021

Abstract

Statistical fault localization is an easily deployed technique for quickly determining candidates for faulty code locations. If a human programmer has to search the fault beyond the top candidate locations, though, more traditional techniques of following dependencies along dynamic slices may be better suited. In a large study of 457 bugs (369 single faults and 88 multiple faults) in 46 open source C programs, we compare the effectiveness of statistical fault localization against dynamic slicing. For single faults, we find that dynamic slicing was eight percentage points more effective than the best performing statistical debugging formula; for 66% of the bugs, dynamic slicing finds the fault earlier than the best performing statistical debugging formula. In our evaluation, dynamic slicing is more effective for programs with single fault, but statistical debugging performs better on multiple faults. Best results, however, are obtained by a *hybrid approach*: If programmers first examine at most the top five most suspicious locations from statistical debugging, and then switch to dynamic slices, on average, they will need to examine 15% (30 lines) of the code. These findings hold for 18 most effective statistical debugging formulas and our results are independent of the number of faults (i.e. single or multiple faults) and error type (i.e. artificial or real errors).

Keywords Software engineering · Software debugging · Software testing · Automated fault localization · Program slicing · Statistical debugging

Communicated by: Hadi Hemmati

✉ Ezekiel Soremekun
ezekiel.soremekun@uni.lu

Lukas Kirschner
s8lukirs@stud.uni-saarland.de

Marcel Böhme
marcel.boehme@acm.org

Andreas Zeller
zeller@cispa.saarland

[1] SnT, University of Luxembourg, Luxembourg, Luxembourg

[2] CISPA – Helmholtz Center for Information Security, Saarbrücken, Germany

[3] Monash University, Melbourne, Australia

1 Introduction

In the past 20 years, the field of *automated fault localization* (AFL) 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 computer-generated patches (Le Goues et al. 2012; Nguyen et al. 2013; Kim et al. 2013; Qi et al. 2014).

The large majority of recent publications on automated fault localization fall into the category of *statistical debugging* (also called *spectrum-based fault localization* (SBFL)), an approach pioneered 18 years ago (Jones et al. 2002; Zheng et al. 2003; Liblit et al. 2005). A recent survey (Wong et al. 2016) lists more than 100 publications on statistical debugging. 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`, used in Jones et al. (2002) to introduce the technique (*see* Fig. 1). The `middle` function computes the middle of three numbers `x, y, z`; Fig. 1 shows its source code as well as statement coverage for 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 them, statistical debugging assigns a *suspiciousness score* to each program statement—a function on the number of times it is (not) executed by passing and failing test cases. The precise function it uses differs for each statistical debugging technique. Since the statement in Line 8 is

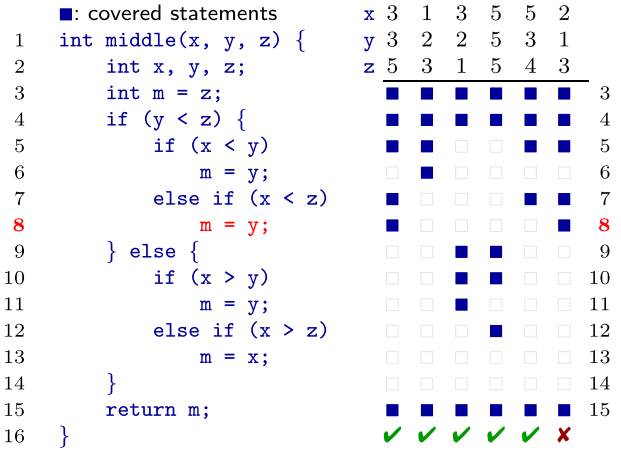

Fig. 1 Statistical debugging illustrated (Jones and Harrold 2005): The `middle` function takes three values and returns that value which is greater than or equals the smallest and less than or equals the biggest value; however, on the input (2, 1, 3), it returns 1 rather than 2. Statistical debugging reports the faulty Line 8 (in **bold red**) as the most suspicious one, since the correlation of its execution with failure is the strongest

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 titled "Programmers use slices when debugging" (Weiser 1982), Mark Weiser introduced the concept of a *program slice* composed of data and control dependencies in the program. Weiser 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, programmers would follow *dynamic* dependencies to find those lines that actually impact the location of interest in the *failing run*. In our example (Fig. 2), they could simply follow the dynamic dependency of Line 15 where the value of `m` is unexpected, and immediately reach the faulty assignment in Line 8. Consequently, on the *example originally introduced to show the effectiveness of statistical debugging* (Fig. 1), the older technique of dynamic slicing is just as effective (*see* Fig. 2).

Thus, we investigate the fault localization effectiveness of *the most effective* statistical debugging formulas against dynamic program slicing. A few researchers have empirically evaluated the fault localization effectiveness of different slicing algorithms (Zang et al. 2007, 2005). However, they did not compare the effectiveness of slicing to that of statistical debugging. To the best of our knowledge, this is the *first empirical study* to evaluate the fault localization effectiveness of program slicing versus (one of) the most effective statistical debugging formulas. This is also one of the *largest empirical studies* of fault localization techniques, evaluating hundreds of faults (707) in C programs.

In this paper, we use four benchmarks with 35 tools, 46 programs and 457 bugs to compare fault localization techniques against each other. This set of bugs comprises of 295 real single faults, 74 injected single faults, and 88 injected multiple faults containing about four

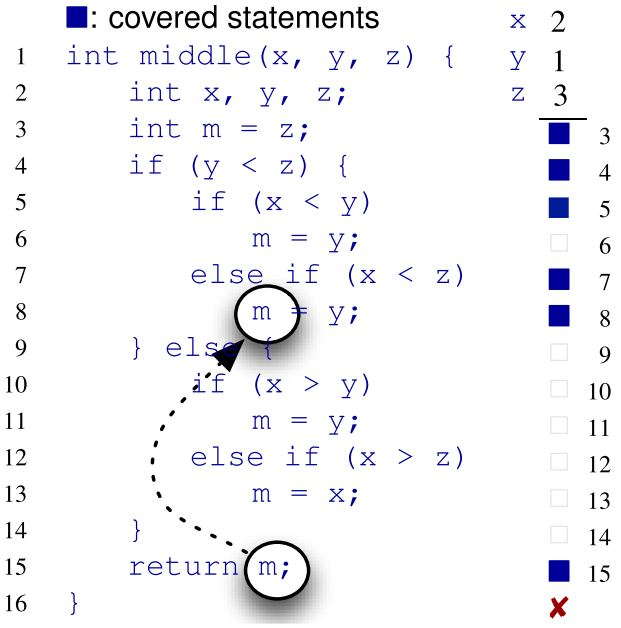

Fig. 2 Dynamic slicing illustrated (Jones and Harrold 2005): The `middle` return value in Line 15 can stem from any of the assignments to `m`, but only those in Lines 3 and 8 are executed in the failing run. Following back the dynamic dependency immediately gets the programmer to Line 8, the faulty one

faults per program, on average. In total, we had 707 program faults. 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 pinpoints the faults, it correctly localizes 33% of faults after inspecting *only the single most suspicious code location*. It outperforms dynamic slicing in the top 5% of the most suspicious locations, by localizing faults in *twice as many* buggy programs as dynamic slicing. In our experiments, looking at only the top 5% of the most suspicious code locations, statistical debugging would reveal faults for 6% of all buggy programs, twice as many as slicing (3% of buggy programs). This result is important for *automatic program repair* (APR) techniques, as the search for possible repairs can only consider a limited set of candidate locations; also, the repair attempt is not necessarily expected to succeed.
2. **If one *must* fix a (single-fault) bug, dynamic slicing is more effective.**[1] In our experiments, dynamic slicing is 62% more likely to find the fault location earlier than statistical debugging, for single faults. In absolute terms, locating faults along dynamic dependencies requires programmers to examine on average 21% of the code (40 LoC); whereas the most effective statistical debugging techniques require 26% (51 LoC). Not only is the average better; the effectiveness of dynamic slicing also has a much lower standard deviation and thus is more predictable. Both features are 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 quicker than if they examine locations whose execution correlates with failure. Moreover, dynamic slicing needs only the failing run, whereas statistical debugging additionally requires multiple similar passing runs. Although dynamic slicing is more effective on single faults, statistical debugging performs better on multiple faults (*see RQ7*).
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* strategy that works best: First consider the top locations of statistical debugging (if applicable), and then proceed along the dynamic slice. In our experiments, *the hybrid approach is significantly more effective than both slicing and statistical debugging.* For most errors (98%), the hybrid approach localizes the fault within the top-20 most suspicious statements, in contrast, both slicing and statistical debugging will localize faults for most errors (98%) after inspecting about five times as many statements (100 LoC). Notably, the hybrid approach is more effective than statistical debugging and dynamic slicing, regardless of the error type (real/artificial) and the number of faults (single/multiple) in a buggy program (*see RQ6 and RQ7, respectively*).

The remainder of this paper is organized as follows. After introducing dynamic slicing and statistical debugging in Section 2, this paper makes the following key contributions:

1. Section 3 presents a *hybrid approach* that merges both dynamic slicing and statistical debugging into a strategy, where the developer switches to slicing after investigating a handful of the most suspicious statements reported by statistical debugging.

[1] In our evaluation, dynamic slicing is more effective than SBFL on single faults. However, other factors such as multiple faults (*see RQ7*), test generation (Yang et al. 2017), test reduction (Yu et al. 2008) and program sizes may influence its effectiveness (*see RQ6 and* Section 6).

2. We describe our evaluation setup (Section 4) and empirically evaluate the fault localization effectiveness of dynamic slicing, statistical debugging and our hybrid approach (*RQ1 to 5 in* Section 5*)*.
3. We conduct an empirical study on the effect of error type and the number of faults on the effectiveness of AFL techniques. We examine the difference between evaluating an AFL technique on *real vs. artificial faults* (*RQ6 in* Section 5*)*, as well as *single vs. multiple faults* (*RQ7 in* Section 5*)*.

In Section 6, we discuss the limitations and threats to the validity of this work. Section 7 and Section 8 present future work and related work, respectively. Finally, Section 9 closes with conclusion and consequences.

The contributions and findings of this paper are important for debugging and repair stakeholders. Programmers, debugging tools and automated program repair (APR) tools need *effective* fault localization techniques, in order to reduce the amount of time and effort spent (automatically) debugging and fixing errors. These findings enable APR tools, debuggers and programmers to be effective and efficient in bug diagnosis and bug fixing.

2 Background

In this section, we provide background on the two main AFL techniques evaluated in this paper, namely *program slicing* and *statistical debugging*.

2.1 Program Slicing

More than three decades ago, Mark Weiser (1982, 1981) 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, developers would identify those program locations that directly influence the variable values or execution of s. This traversal continues transitively, until the root cause of the failure (i.e., the *fault*) is found. This procedure allows developers to investigate those parts of the program involved in the infected information-flow in reversed order towards the location where the failure is first observed.

2.2 Static Slicing

Weiser developed *program slicing* as the first automated fault localization technique (Weiser 1982). A programmer 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* (Weiser 1981; Tip 1995) 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 (Binkley et al. 2007).

2.3 Dynamic, Relevant, and Execution Slicing

A *dynamic program slice* (Korel and Laski 1988; Agrawal and Horgan 1990) 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, but the nodes are the statement *instances* in the execution trace $\pi(t)$. The DDG contains a separate node for each occurrence of a statement in $\pi(t)$ with outgoing dependence edges to only those statement instances on which this statement instance depends in $\pi(t)$ (Agrawal and Horgan 1990). However, an error is not only explained by the actual information-flow towards C. It is important to also investigate statements that could have contributed towards an alternative, potentially correct information-flow.

The *relevant slice* (Agrawal et al. 1993; Gyimóthy et al. 1999) computed for a failing input t subsumes the dynamic slice for t and 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. (2013) 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* (Agrawal et al. 1990; Korel and Laski 1988) 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 (1) *easier to compute* than dynamic slices (static analysis), (2) *significantly smaller* than the static slice, and still (3) *as "complete"* as the relevant slice. In summary, *dynamic slice* $\subseteq$ *relevant slice* $\subseteq$ *approximate dynamic slice* $\subseteq$ *static slice*.

Figure 3a 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 `x` = 2, `y` = 1, and `z` = 3. In this example, the approximate dynamic slice matches exactly the dynamic slice. For our evaluation, we implemented approximate dynamic slicing, and in our evaluation results and discussions *we refer to approximate dynamic slicing as "dynamic slicing"*.

2.4 Statistical Debugging

Almost two decades ago, Jones et al. (2002) introduced the first statistical debugging technique—TARANTULA, quickly followed by Zheng et al. (2003) and Liblit et al. (2005). The main idea of *statistical debugging* is to associate the execution of a particular program element with the occurrence 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. An important property of statistical debugging is that apart from measuring coverage, it requires no specific static or dynamic program analysis. This made it easy to implement and deploy, in particular as part of several *automated program repair techniques* (Le Goues et al. 2012; Nguyen et al. 2013; Kim et al. 2013; Qi et al. 2014), 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.

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 in the example. Notice that only twelve (12) lines are actually executable. Evidently, in this

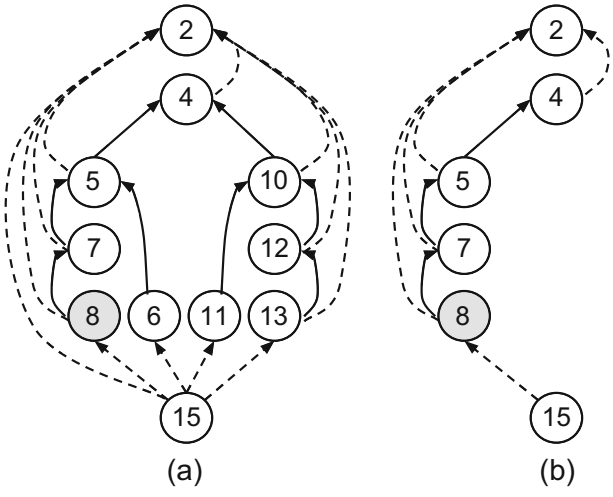

Fig. 3 Slicing Example: Nodes are statements in each line of the `middle` program (see Fig. 1). Control-dependencies are shown as dashed lines while data dependencies are shown as concrete lines

```
int middle(x, y, z) {
    int x, y, z;
    int m = z;
    if (y < z) {
        if (x < y)
            m = y;
        else if (x < z)
            m = y;
    } else {
        if (x > y)
            m = y;
        else if (x > z)
            m = x;
    }
    return m;
}
```

Tarantula	Ochiai	Naish2
0.500	0.408	0.167
0.500	0.408	0.167
0.500	0.408	0.167
0.625	0.500	0.500
0.000	0.000	−0.167
0.714	0.578	0.667
0.833	**0.707**	**0.833**
0.000	0.000	−0.333
0.000	0.000	−0.333
0.000	0.000	−0.167
0.000	0.000	−0.167
0.000	0.000	0.000
0.000	0.000	0.000
0.500	0.408	0.167

Fig. 4 Statistical Fault Localization Example: Scores for the faulty line 8 are in **bold red**

example from Jones and Harrold (2005), the faulty statement is also the most suspicious for these three statistical fault localization techniques.[2]

In this paper, we focus on four sets of measures consisting of 18 statistical fault localization formulas; namely seven human-generated optimal measures, three most popular measures, four Genetic Programming (GP) evolved measures and four measures targeted at *single bug optimality*.

1. **Human-generated measures:** The first set of measures includes two `DStar` (D^*) formulas and five formulas which have been theoretically proven to be optimal and found to be the most effective in existing studies (Xie et al. 2013b). These formulas include `Wong1`, `Russel_Rao`, `Binary`, `Naish1` and `Naish2` (Wong et al. 2007, 2012, 2013; Russel et al. 1940; Naish et al. 2011). For the `DStar` algorithm, we have selected "*star*" (*) values two and three (i.e., $D^* = \{D^2, D^3\}$) which have been demonstrated to be the most effective values for single and multiple faults, respectively (Wong et al. 2012, 2013). The other five measures were selected in a theoretical evaluation of over 50 formulas and recommended as the only optimal formulas to be applied for statistical fault localization (Xie et al. 2013a, b).
2. **Popular measures:** These measures are the most popular statistical fault localization measures, `Tarantula`, `Ochiai`, and `Jaccard` (Jones et al. 2002; Abreu et al. 2006, 2007; Chen et al. 2002). They have been used in recent automated program repair (APR) techniques and have been shown to improve the effectiveness of program repair (Nguyen et al. 2013; Assiri and Bieman 2017).
3. **Genetic Programming (GP) evolved measures:** These measures are GP-evolved formulas, which have been found to be human-competitive (comparable to human-generated measures) and theoretically maximal (i.e., the best performing measures), namely `GP02`, `GP03`, `GP13` and `GP19` (Yoo 2012).

[2]The scores for the faulty statement in Line 8 are $\texttt{tarantula}(s_8) = \frac{1}{1} / \left(\frac{1}{1} + \frac{1}{5}\right)$, $\texttt{ochiai}(s_8) = \frac{1}{\sqrt{1(1+1)}}$, and $\texttt{naish2}(s_8) = 1 - \frac{1}{1+4+1}$.

4. **Single Bug Optimal measures:** These statistical formulas are optimized for programs containing a single bug, based on the observation that if a program contains only a single bug, then all failing traces cover that bug (Naish and Lee 2013). These measures have been empirically demonstrated to be optimal in a large-scale comparison of 157 measures. The single bug optimal measures in our study include `m9185`, `Kulczynski2` `LexOchiai` and `Pattern-Similarity`. In particular, `LexOchiai` and `Pattern-Similarity` measures perform best overall (Landsberg 2016; Landsberg et al. 2015).

3 A Hybrid Approach

Even though dynamic slicing is generally more effective than statistical debugging, we observe that statistical debugging can be highly effective for some bugs, especially when inspecting only the top most suspicious statements. For instance, statistical debugging can pinpoint a single faulty statement as the most suspicious statement for about 40% of the errors in `IntroClass` and `SIR`, i.e. a developer can find a faulty statement after inspecting only one suspicious statement (*see* Fig. 8). This is further illustrated by the clustering of some points in the rightmost corner below the diagonal line of the comparison charts (*see* Fig. 7).

In this paper, we assume that a programmer in the end *has* to fix a bug, and a viable "alternative" method is following the dependencies by (dynamic) slicing. To this end, we investigate a *hybrid* fault localization approach which leverages the strengths of both dynamic slicing and statistical debugging. 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 the *top most suspicious statements* (e.g. top five statements) before it reports the statements in the dynamic slice computed w.r.t. the symptomatic statement.

The concept of examining only the top most suspicious statements is also backed by user studies on statistical fault localization. In a recent survey (Kochhar et al. 2016), Kochhar et al. found that three quarter of surveyed practitioners would investigate *no more than the top-5 ranked statements*—which should contain the faulty statement at least three out of four times—before switching to alternative methods. This is also confirmed by the study of Parnin and Orso (2011), who observed 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. This transition is exactly what the hybrid approach provides.

Specifically, the hybrid approach proceeds in two phases. In the first phase, it reports the top N (e.g. $N = 5$) most suspicious statements, obtained from the ordinal ranking[3] of a statistical fault localization technique. Then, if the fault is not found, it proceeds to the second phase where it reports the symptom's dynamic backward dependencies. In the second phase, we only report statements that have not already been reported in the first phase.[4]

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

[4]This is to avoid duplication of inspected statements, i.e. avoid double inspection.

4 Evaluation Setup

Let us evaluate the effectiveness of all three fault localization techniques and the influence of the number of faults and error type on the effectiveness of these AFL techniques. Specifically, we ask the following research questions:

- **RQ1: Effectiveness of Dynamic Slicing**: How effective is dynamic slicing in fault localization, i.e. localizing fault locations in buggy programs?
- **RQ2: Effectiveness of Statistical Debugging**: Which statistical formula is the most effective at fault localization?
- **RQ3: Comparing Statistical Debugging and Dynamic Slicing**: How effective is the most effective statistical formula in comparison to dynamic slicing?
- **RQ4: Sensitiveness of the Hybrid Approach**: How many suspicious statements (reported by statistical debugging, i.e. Kulczynski2) should a tool or developer inspect before switching to slicing?
- **RQ5: Effectiveness of the Hybrid Approach**: Which technique is the most effective in fault localization? Which technique is more likely to find fault locations earlier?
- **RQ6: Real Errors vs. Artificial Errors**: Does the type of error influence the effectiveness of AFL techniques? Is there a difference between evaluating an AFL technique on real or artificial errors?
- **RQ7: Single Fault vs Multiple Faults**: What is the effect of the number of faults on the effectiveness of AFL techniques? Is there a difference between evaluating an AFL technique on single or multiple fault(s)?

In this paper, we evaluate the performance of statistical debugging, dynamic slicing and the hybrid approach in the framework of Steimann et al. (2013) where we fix the granularity of fault localization at *statement level* and the fault localization mode at *one-at-a-time* (except for multiple faults in RQ7). In this setting with real errors and real test suites, the provided test suites *may not be coverage adequate*, e.g. they may not execute all program statements. Fault localization effectiveness is evaluated as *relative wasted effort* based on the ranking of units in the order they are suggested to be examined (see Section 4.4 for more details).

4.1 Implementation

Let us provide implementation details for each AFL technique in this paper.

4.1.1 Dynamic Slicing Implementation

The approximate dynamic slice is computed using `Frama-C`,[5] `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

[5] http://frama-c.com/

approximate dynamic slice. We use the Python libraries `pygraphviz`,[6] `networkx`,[7] and `matplotlib`[8] to process the DOT files and compute the *score* for the approximate dynamic slice.

4.1.2 Statistical Debugging Implementation

The statistical debugging tool was implemented using two bash scripts with several standard command line tools, notably `gcov`,[9] `git-diff`[10] and `gdb`.[11] 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. Some errors were thus excluded from the evaluation. 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 some cases, no coverage information could be generated due to infinite recursion. `Gcov` also gives the number of *executable* statements in the buggy program (i.e., $|P|$).[12] Finally, our Python implementation of the scores is used to compute the fault localization effectiveness.

4.1.3 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. Given the unranked suspiciousness score of every executable statement in the program, the hybrid fault localizer performs an ordinal ranking of all statements. It then determines the proportion of the top N rank of suspicious statements, based on the N value of the hybrid approach. For instance, a hybrid approach with $N = 5$ takes the five topmost suspicious statements. Then, it determines the highest ranked faulty statement in the rank of all suspicious statements. If the faulty statement is in the top N suspicious positions (e.g. third position), then it reports the number of statements in the top ranked positions up till the faulty statement, as a proportion of all executable program statements.

In the case that the suspicious statement is not in the top N suspicious positions (e.g. seventh position), then it proceeds to slicing and reports the cardinality of the set union of all N top ranked statements and the number of inspected statements in the slice before the first faulty statement.

[6] https://pygraphviz.github.io/
[7] https://networkx.github.io/
[8] http://matplotlib.org/
[9] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[10] https://git-scm.com/docs/git-diff
[11] https://www.gnu.org/software/gdb/documentation/
[12] The executable statements refers to statements for which coverage information are obtainable by `Gcov`, in particular, all program statements except spaces, blanks and comments.

4.2 Metrics and Measures

Odds Ratio ψ. To establish the superiority of one technique A over another technique B, it is common to measure the effect size of A w.r.t. B. A standard measure of effect size and widely used is the *odds ratio* (Grissom and Kim 2005). It "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" (Grissom and Kim 2005). In our case, let "A is *successful*" mean that fault localization technique A is more effective than fault localization technique B and let a be the number of successes for A, 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 five 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.

The Mann-Whitney U-test is used to show whether there is a statistical difference between two techniques (Mann and Whitney 1947). In general, it is a non-parametric test of the null hypothesis that two samples come from the same population against an alternative hypothesis, especially that a particular population tends to have larger values than the other. Unlike the t-test it does not require the assumption that the data is normally distributed. More specifically, it shows whether the difference in performance of two techniques is actually statistically significant.

A *cumulative frequency curve* is a running total of frequencies. We use such curves to show the percentage of errors that require examining up to a certain number of program locations. The number of code 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.

4.3 Objects of Empirical Analysis

Programs and Bugs We evaluated each fault localization technique using 45 C programs containing hundreds of (369) errors and thousands of (9012) failing tests (*cf.* Table 1). These programs were collected from four benchmarks, in particular, three benchmarks containing real world errors, namely IntroClass, Codeflaws and CoREBench, and one benchmark with artificial faults, namely the Software-artifact Infrastructure Repository (SIR). We selected these benchmarks of C programs to obtain a large variety of bugs and programs. Each benchmark contains a unique set of programs containing errors introduced from different sources such as developers, students, programming competitions and fault seeding (e.g. via code mutation). These large set of bugs allows us to rigorously evaluate each fault localization technique. The following briefly describes each benchmark used in our evaluation:

1. *Software-artifact Infrastructure Repository (SIR)* is a repository designed for the evaluation of program analysis and software testing techniques using controlled experimentation (Hutchins et al. 1994). It contains small C programs, with seeded errors and

Table 1 Details of subject programs

Benchmark (error type)	Tool (program)	Avg. size (LoC)	#Errors	#Fail. tests	#Pass. tests
SIR	tcas	65.1	37	1356	58140
(Artificial)	print_tokens	199	3	184	12206
	print_tokens2	199.5	8	2031	30889
	tot_info	125	18	1528	17408
	schedule	160.5	4	690	9910
	schedule2	139.2	4	116	10724
IntroClass	checksum	11.3	3	7	41
(Real; Students)	digits	17.4	3	16	32
	grade	16.1	8	30	114
	median	13.5	2	8	18
	smallest	13.2	2	16	16
	syllables	11.6	2	12	20
Codeflaws	WTLW (71A)	10.3	11	60	61
(Real; Competitions)	HQ9+ (133A)	10.9	18	270	1260
	AG (144A)	24.5	13	302	202
	IB (478A)	8.6	20	31	329
	TN (535A)	61.2	9	118	778
	Exam (534A)	17.5	12	108	68
	Holidays (670A)	12.8	9	662	1118
	DC (495A)	14.5	13	96	279
	VBT(336A)	13.6	14	108	309
	PP(509B)	21.2	16	84	98
	DHHF (515B)	29.5	15	127	707
	HVW2 (143A)	17.5	16	124	707
	Ball Game (46A)	10	8	114	148
	WE (31A)	14.4	14	187	200
	LM (146B)	29.5	11	116	355
	SG (570B)	7.5	11	69	531
	WD (168A)	7.7	9	132	254
	Football (417C)	13.2	13	64	352
	MS (218A)	16.5	10	156	156
	Joysticks (651A)	12.6	8	66	246
CoREBench	core. (cut)	306	4	4	6
(Real; Developers)	core. (rm)	110	1	1	63
	core. (ls)	1605.5	2	2	73
	core. (du)	315	1	1	28
	core. (seq)	219.7	3	3	5
	core. (expr)	321	1	1	1
	core. (copy)	897	1	1	59
	find (parser)	119.3	3	3	286
	find (ftsfind)	211.5	2	2	183
	find (pred)	825	2	2	235

Table 1 (continued)

Benchmark (error type)	Tool (program)	Avg. size (LoC)	#Errors	#Fail. tests	#Pass. tests
	`grep (dfasearch)`	181.5	2	2	46
	`grep (savedir)`	64	1	1	15
	`grep (kwsearch)`	77	2	2	46
	`grep (main)`	853.5	2	2	45
Total	35 (46)		369	9012	148767

test suites containing thousands of failing tests. In particular, this benchmark allows for the controlled evaluation of the effects of large test suites on debugging activities.

2. *IntroClass* is a collection of small programs written by undergraduate students in a programming course (Le Goues et al. 2015). It contains six C programs, each with tens of instructor-written test suites. This benchmark allows for the evaluation of factors that affect debugging in a development scenario, especially for novice developers.
3. *Codeflaws* is a collection of programs from online programming competitions held on Codeforces.[13] These programs were collected for the comprehensive evaluation of debugging tools using different types of errors. It contains 3902 errors classified across 40 defect classes in total (Tan et al. 2017). In particular, this benchmark allows for the evaluation of fault localization techniques on different defect types.
4. *CoREBench* 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 (Böhme and Roychoudhury 2014).[14] 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. CoREBench allows for the evaluation of fault localization techniques on real world errors (unintentionally) introduced by developers. It has been used in several debugging studies, including a study that investigates how developers debug and fix real faults (Böhme et al. 2017).

Table 1 lists all the programs and bugs investigated in our study. We use six programs each from the SIR and IntroClass benchmarks. This includes `tcas`—this program is *the* most well-studied subject according to a recent survey on fault localization (Wong et al. 2016). We selected 20 programming competitions from Codeflaws, including popular and difficult contests, such as "Tavas and Nafas (535A)" and "Lucky Mask (146B)". From CoREBench, we used three projects, namely the `Coreutils`, `Grep` and `Find` project. Notably, all projects in CoREBench come from the GNU open source C programs, in particular, these three projects contain a total of 103 tools. Due to code modularity, the program size for a single tool (e.g. `cut` in `coreutils`) contains a few hundred LoC (about 306 LoC), however, the entire code base for CoREBench is fairly large. For instance, `Coreutils`, `Grep` and `Find` have 83k, 18k and 11k LoC, respectively (Böhme and Roychoudhury 2014). For each benchmark, we exempted programs where `Frama-C` *could not* construct the Program Dependence Graph (PDG). For instance, because it cannot handle

[13] https://codeforces.com/

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

Table 2 Details of multiple faults

Benchmark (error type)	Tool	# Buggy programs	#Faults	#Failing tests	#Passing tests
SIR-MULT (Mutated)	tcas	37	144	19973	39523
	print_tokens	3	11	12074	316
	print_tokens2	8	28	27630	5290
	tot_info	18	64	16667	2269
	schedule	4	17	9673	927
	schedule2	4	16	8616	2224
IntroClass -MULT (Mutated)	checksum	1	4	15	1
	digits	2	7	30	2
	grade	5	22	67	23
	median	2	8	13	13
	smallest	1	4	8	8
	syllables	3	13	34	14
Total		88	338	94800	50610

some recursive or variadic method calls. In addition, we excluded an error if no coverage information could be generated (e.g., infinite loops) or the faulty statement could not be identified (e.g., *omission faults* where the patch only added statements).

Single Faults For our evaluation (all RQs except RQ7), we used buggy programs collected from four well-known benchmarks, where programs contained only a single fault. To determine single faults in our bug dataset, for each program, we executed all tests available for a project on the fixed version of the program, in order to determine if there are any failing test cases that are unrelated to the bug at hand. Our evaluation revealed that our dataset contained mostly single bugs (368/369 = 99.7%). Almost all buggy program versions had exactly one fault, except for a single program—Codeflaws version DC 495A. For all benchmarks, only this program contained multiple faults, i.e. more than one fault. This distribution of single faults portrays the high prevalence of single faults and single-fault fixes in the wild (Perez et al. 2017).

Multiple Faults To evaluate the effectiveness of all three fault localization techniques on multiple faults (*see* RQ7), we automatically curated a set of multiple faults using *mutation-based fault injection*. This is in line with the evaluation of multiple faults in previous works (Abreu et al. 2009b; Zheng et al. 2006; DiGiuseppe and Jones 2011; Wong et al. 2013; Wong et al. 2012).[15] We automatically mutated the original passing version of each program until we have a buggy version containing between three to five faults. In particular, we performed logical and arithmetic operator mutation on each passing version of the programs contained in the SIR and IntroClass benchmarks. Table 2 provides details of the buggy programs with multiple faults, the number of faults, as well as the number of failing and passing test cases. For each fault contained in the resulting program, we store the failing

[15]To the best of our knowledge, there is no known benchmark of real-world programs containing multiple faults.

test case(s) that expose the bug, as well as the corresponding patches for each fault and all faults. In total, we collected 88 programs with multiple faults containing 338 injected faults, in total. Each program in this dataset contained about four unique faults, on average. Specifically, we collected 74 and 14 programs from the `SIR` and `IntroClass` benchmarks, and injected a total of 280 and 58 faults in each benchmark, respectively. The programs containing multiple faults are called `SIR-MULT` and `IntroClass-MULT`, respectively (*see* Table 2).

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 (2003) recommend identifying as faulty statements those that need to be changed to derive the (correct) program that does not contain the error. For each error, only patched statements are considered faulty. All bugs in our corpus are patched with at least one statement changed in the buggy program, all *omission bugs* are exempted. Omission bugs require special handling since they are quite difficult to curate, localize and fix. Collecting patches and fault locations for omission bugs is difficult because their patches are similar to the implementation of new features. A faulty code location is unclear for omission bugs (in the failing commit), this makes them even more difficult to evaluate for typical AFL techniques, including statistical debugging and dynamic slicing (Lin et al. 2018).

Slicing Criterion All aspects of dynamic slicing can be fully automated. To this end, as the 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 the slicing criterion. In our implementation, the slicing criterion is automatically selected by a bash script running `gdb`.

Passing and Failing Test Cases All programs in our dataset come with an extensive test suite which checks corner cases and that previously fixed errors do not re-emerge. For statistical debugging, we execute each of these (passing) test cases individually to collect coverage information. For dynamic slicing, we perform slicing for each failing test case.

In summary, for our automated evaluation, we used 457 errors in dozens of programs from four well-known benchmarks (*see* Tables 1 and 2). Our corpus contained 46 different programs in 35 software tools. Each faulty program in our corpus had about 11 bugs, 257 failing test cases and thousands (4250) of passing test cases, on average. For single faults, we have 295 real faults and 74 injected faults. Meanwhile, we have 88 buggy programs containing multiple faults, each program contains about four faults, on average.

4.4 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 proportion 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 a study of Böhme and Roychoudhury, only about 10% of failures were caused by a single statement, while there is a long tail of failures that are substantially more complex (Böhme and Roychoudhury 2014). 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. (2016) motivates this measure of effectiveness and presents an overview of other measures.

4.4.1 General Measures

Ranking All three fault localization techniques presented in this paper produce a ranking. The developer starts examining the highest ranked statement 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 all 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 one.[16]

Multiple Statements, Same Rank In most cases there are several statements that have the same rank as the faulty statement. For all our evaluations, we employ ordinal ranking, in order to effectively determine the top N most suspicious statements for each technique. This is necessary to evaluate the fault localization effectiveness of each technique, if a developer is only willing to inspect N most suspicious statements (Kochhar et al. 2016). In ordinal ranking, lines with the same score are re-ranked by line numbers.[17] This is in agreement with evaluations of fault localization techniques in previous work (Wong et al. 2016; Pearson et al. 2017; Kochhar et al. 2016).

Multiple Faults, Expense Score For *multiple faults*, we measure fault localization effectiveness using the *expense score* (Yu et al. 2008). The expense score is the percentage of the program (statements) that must be examined to find the *first fault*, in particular, the *first faulty statement in the first localized fault*, using the ranking given by the fault localization technique. It is similar to the score employed for single faults, and it has been employed in previous evaluations of multiple faults, such as Wong et al. (2012, 2013). Formally:

$$expense\ score = \frac{|S|}{|P|} * 100$$

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

[16] Note that all executable program statements are ranked in the suspiciousness rank, executable statements that are not contained in the dynamic slice are ranked lowest.

[17] Ranking ties are broken in ascending order, i.e. if both lines 10 and 50 have the same score, then line number 10 is ranked above line number 50.

statement, for the first localized fault. The assumption is that it is the first fault that the developer would begin fixing, thus, finding the first statement suffices for the diagnosis of all faults (Yu et al. 2008). In our evaluation of multiple faults, the fault localization effectiveness *score* is computed similarly to single faults as

$$score_{\text{mult}} = 1 - (expense\ score/100)$$

4.4.2 Dynamic Slicing Effectiveness

We define the effectiveness of *approximate dynamic slicing*, the $score_{ads}$ according to Renieris and Reiss (2003) as follows. Given a failing test case t, the symptomatic statement c, let P be the set of all statements in the program, 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.

In the approximate dynamic slice in our motivating example (Fig. 3), we have $score_{ads} = 1 - \frac{1}{12} = 0.92$. 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*.[18] Statements s_5 and s_4 are ranked fourth and fifth, respectively, while the remaining, not executed (but executable) statements are ranked 12th.

4.4.3 Statistical Debugging Effectiveness

We define the effectiveness of a *statistical fault localization* technique, the $score_{sfl}$ as follows. Given the ordinal ranking of program statements in program 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 and P is the set of all statements in the program. Then,

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

Note that $score_{sfl} = 1 - EXAM\text{-}score$ where the well-known *EXAM-score* (Jones and Harrold 2005; Abreu et al. 2009a) 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

[18] In this case, when several statements have the same rank as the faulty statement, we made the conservative assumption that a developer finds the faulty statement among other statements with the same rank.

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.

4.4.4 Hybrid Approach Effectiveness

We define the effectiveness of the hybrid approach, the $score_{hyb}$ as follows. Let R be the set of faulty statements, H be the N most suspicious statements—sorted first by suspiciousness score and then by line numbers and P is the set of all statements in the program. 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 Fig. 1 since the fault is amongst the N most suspicious statements.

4.5 Infrastructure

We performed the experiments on a virtual machine (VM) running Arch Linux. The VM was running on a Dell Precision 7510 with a 2.7 GHz Intel Core i7-6820hq CPU and 32 GB of main memory.

5 Evaluation Results

Let us discuss the results of our evaluation and their implications. All research questions (RQs) are evaluated using single faults (i.e., RQ1 to RQ6), except for RQ7, which is evaluated on both single and multiple faults.

5.1 RQ1: Effectiveness of Dynamic Slicing

How effective is dynamic slicing in fault localization? To investigate the fault localization effectiveness of dynamic slicing, we examined the proportion of statements a developer would *not need to* inspect after locating the faulty statement (*score* in Table 3). Then, we examine the percentage of errors for which a developer can effectively locate the faulty statement, if she inspects only N most suspicious statements reported by dynamic slicing for $N \in \{5, 10, 20, 30\}$ (*% Errors Localized* in Table 3).

Overall, a single faulty statement is ranked within the first quarter of the most suspicious program statements reported by dynamic slicing, on average (*cf.* Table 3). This implies that a developer (using dynamic slicing) inspects 21% (about 40 LoC) of the executable program

Table 3 Effectiveness of dynamic slicing on single faults

Benchmark	Score	% Errors localized if developer inspects N most suspicious LoC			
		5	10	20	30
IntroClass	0.83	70.00	100	100	100
Codeflaws	0.78	75.30	92.71	98.79	100
CoREBench	0.85	18.52	18.52	29.63	40.74
Real	0.79	69.73	86.39	92.52	94.56
Artificial (SIR)	0.79	32.43	44.59	55.41	60.81
Avg. (Bugs)	0.774	62.23	77.99	85.05	87.77

statements before locating the fault, on average (i.e., equal to $1 - score$). This performance was independent of the source or type of the errors (i.e., real or seeded errors). Dynamic slicing was particularly highly effective in locating faults for errors in CoREBench and errors in IntroClass, where it ranks the faulty statements within the top 15% (81 LoC) and 17% (3 LoC) of the program statements, respectively (*cf.* Table 3).

For all programs, dynamic slicing reports the faulty statement within the top 21% (40 LoC) of the most suspicious statements, on average.

A developer or tool using dynamic slicing will locate the faulty statement after inspecting a handful of suspicious statements. In our evaluation, for most errors, the faulty statement can be identified after inspecting only five to ten most suspicious statements reported by dynamic slicing. Specifically, the faulty statement is ranked within the top five to ten most suspicious statements for 62% to 78% of all errors, respectively (*cf.* Table 3). Notably, a developer will locate the faulty statement for 55% of artificial errors and 92% of real errors if she inspects the top 20 most suspicious statements. Overall, most programs (85%) can be debugged by inspecting the top 30% (58 LoC, on average) of the statements reported by dynamic slicing. These results demonstrate the high effectiveness of dynamic slicing in fault localization.

Dynamic slicing reports a single faulty statement within the top 5–10 most suspicious statements for most errors (62% to 78%, respectively).

5.2 RQ2: Effectiveness of Statistical Debugging

Which statistical formula is the most effective at fault localization? First, we investigate the effectiveness of 18 statistical formulas using four benchmarks containing 369 errors (*cf.* Table 1). To determine the most effective statistical formula, for each formula, we examined the proportion of statements a developer would *not need* to inspect after locating a single faulty statement (*score* in Table 4). Figure 5a and b further illustrate the effectiveness

Table 4 Effectiveness of statistical debugging on single faults

SBFL family	Formula	SIR	Intro Class	Code flaws	Core bench	Average (Bugs)	Average (Prog.)
Popular	Tarantula	0.78	**0.76**	**0.70**	**0.79**	**0.709**	0.732
	Ochiai	**0.83**	**0.76**	0.69	**0.79**	**0.709**	**0.735**
	Jaccard	0.80	**0.76**	0.69	**0.79**	0.702	0.728
Human Generated	Naish_1	**0.83**	**0.74**	**0.69**	0.79	**0.710**	**0.733**
	Naish_2	0.81	**0.74**	**0.69**	0.79	0.709	0.731
	Russel_Rao	0.67	0.59	0.57	0.77	0.602	0.611
	Binary	0.69	0.59	0.57	0.77	0.603	0.614
	Wong_1	0.67	0.59	0.57	0.77	0.602	0.611
	D^2	0.73	0.62	0.56	**0.80**	0.598	0.618
	D^3	0.75	0.62	0.56	**0.80**	0.601	0.622
GP Evolved	GP_02	0.75	0.72	0.66	0.69	0.668	0.688
	GP_03	0.77	0.68	0.63	0.63	0.643	0.663
	GP_13	**0.81**	**0.74**	**0.69**	**0.79**	**0.709**	**0.731**
	GP_19	0.56	0.69	0.65	0.75	0.631	0.649
Single Bug Optimal	PattSim_2	**0.85**	0.68	0.69	0.76	0.705	0.721
	lex_Ochiai	0.83	0.74	0.69	**0.79**	0.710	0.733
	m9185	0.83	0.74	**0.70**	**0.79**	**0.715**	0.735
	Kulczynski2	0.83	**0.76**	**0.70**	**0.79**	0.713	**0.737**

Best scores for each (sub)category are in **bold**; higher scores are better. For instance, Kulczynski2 is the best performing (single bug optimal) formula for *all programs (0.737)*, on average

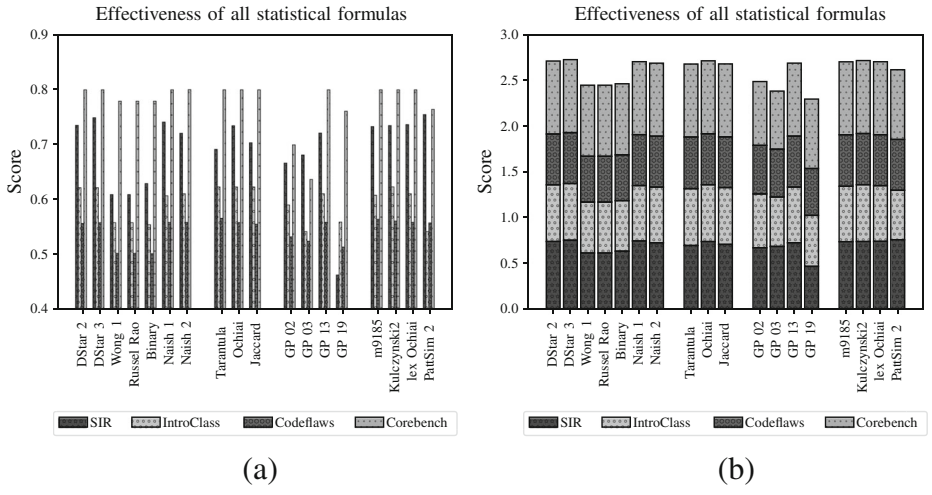

Fig. 5 Effectiveness of each SBFL formula on Single Faults; Results are grouped into bars for each family showing **a** the performance of each SBFL formula on each benchmark and **b** the cumulative results for all benchmarks using stacked bars

of the SBFL formulas. Then, for the best performing statistical formula, we inspected the percentage of errors for which a developer can effectively locate the faulty statement, if she inspects only N most suspicious statements for $N \in \{5, 10, 20, 30\}$ (*% Errors Localized* in Table 5).

Overall, the *single bug optimal* formulas are *the most effective family of statistical formulas*, they are the best performing formulas across all errors and programs. In particular, on average, `PattSim2` performed best for injected errors (i.e. `SIR`), while `Kulczynski2` outperformed all other formulas for real errors, especially for `IntroClass` (*cf.* Table 4). **Bold** values in Table 4 indicate the best performing formula for each family and (sub)category. For instance, `Kulczynski2` is the best performing (single bug optimal) formula for *all programs (0.737)*. The performance of single-bug optimal formulas supports the results obtained in previous works (Landsberg 2016). This family of statistical formulas are particularly effective because they are optimized for programs containing a single bug; based on the observation that if a program contains only a single bug, then all failing traces cover that bug (Naish and Lee 2013).

The `single bug optimal` *statistical formulas outperformed all other SBFL formulas, for both injected and real errors, on average.*

The *most effective statistical formula is* `Kulczynski2`, it outperformed all other formulas in our evaluation (*see* Table 4 *and* Fig. 5a and b). The most effective statistical formula for each family are `Ochiai`, `Naish_1`, `GP_13` and `Kulczynski2` for the *popular*, *human-generated*, *genetically-evolved* and *single bug optimal* families, respectively. Figure 6a and b compares the performance of the most effective formula in each family. For instance, in the *popular* statistical family, `Ochiai` is the best performing formula, both for *all errors (0.709)* and *all programs (0.735)* (*cf.* Table 4). Meanwhile, in the single bug optimal family, `Kulczynski2` is the best performing formula for *all programs (0.737)* (*cf.* Table 4).

Indeed, a developer using `Kulczynski2` will inspect the least number of suspicious program statements before finding the faulty statement. On average, `Kulczynski2` required a developer to inspect about 26% (51 LoC) of the program code before finding the faulty statement. Among all statistical formulas, it has the highest suspiciousness rank for

Table 5 Effectiveness of `Kulczynski2` (i.e., the most effective statistical formula) on Single faults

Benchmark	Score	% Errors Localized if developer inspects **N** most suspicious LoC			
		5	10	20	30
IntroClass	0.76	80.00	85.00	100	100
Codeflaws	0.69	64.37	86.23	97.57	99.19
CoREBench	0.79	22.22	25.93	37.04	48.15
Real	0.72	61.56	80.61	92.18	94.56
Artificial (SIR)	0.83	35.14	41.89	68.92	71.62
Avg. (Bugs)	0.713	56.25	72.83	87.50	89.95

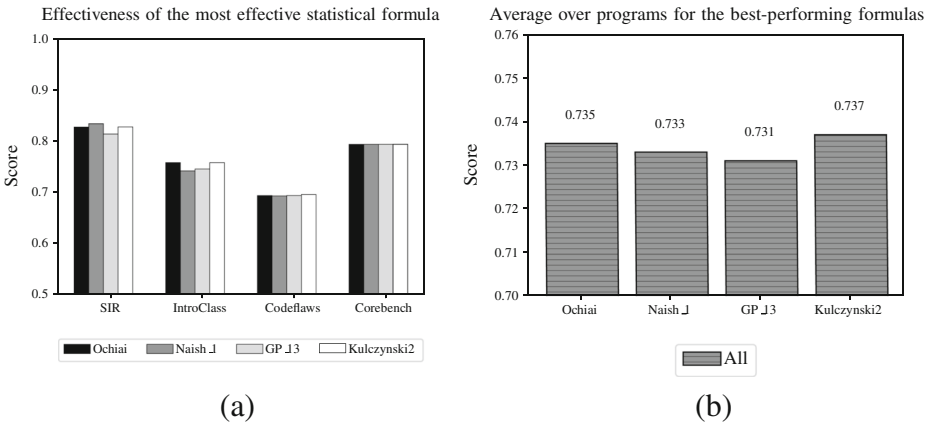

Fig. 6 Effectiveness of the most effective statistical debugging formula in **a** each family (bars are grouped by benchmarks), and **b** the overall average (i.e., mean) for all benchmarks

40% (14 out of 35) of the programs and 72% (265 out of 369) of all errors. It is also the most effective statistical formula for localizing real errors.[19]

> `Kulczynski2` *is the most effective statistical formula, requiring a developer to inspect 26% of code (51 LoC) before finding the fault, on average.*

A tool or developer using `Kulczynski2` will locate the faulty statement after inspecting five to ten most suspicious statements. The faulty statement is ranked within the top five to ten most suspicious statements for most errors, i.e. 56% to 73% of all errors, respectively (*cf.* Table 5). Overall, most programs (60%) can be debugged by inspecting the top 30% (58 LoC) of the suspicious statements reported by `Kulczynski2`.

> `Kulczynski2` *reports the faulty statements within the top 5–10 most suspicious statements for 56% to 72% of all errors, respectively.*

Is the difference in the performance of `Kulczynski2` *statistically significant, in comparison to the best performing formula for each statistical debugging family?* In our evaluation, the difference in the performance of `Kulczynski2` (i.e. the best performing formula) is not statistically significant. Table 6 highlights the statistical tests comparing `Kulczynski2` to the best performing statistical formula in each family, i.e. `Kulczynski2` vs. {`Ochiai`, `Naish1`, `GP13`}. Notably, the performance of `Kulczynski2` is not statistically significant, in comparison to the best statistical formula for each family. This is evident from the fact that the *odds ratio is less than one* ($\psi < 1$) *for all test comparisons* (*see* Table 6). This suggests that `Kulczynski2` has no statistically

[19]Further evaluations (on single faults) in this paper use `Kulczynski2` as the default "*statistical debugging*" formula.

Table 6 Statistical Tests for the most effective Statistical Debugging Formulas; *Odds ratio* ψ (all ratios are statistically significant *Mann-Whitney U-test* < 0.05 for all tests)

Benchmark	Odds ratio ψ (Mann Whitney test score U)		
	`Kulczynski2` vs. `Ochiai`	`Kulczynski2` vs. `Naish1`	`Kulczynski2` vs. `GP_13`
`SIR`	0.2985 (0.0002)	0.0004 (0)	0.0004 (0)
`IntroClass`	0.0006 (0)	0.0183 (0)	0.0059 (0)
`Codeflaws`	0.0013 (0)	0.0005 (0)	0.0002 (0)
`CoREBench`	0.0003 (0)	0.0003 (0)	0.0003 (0)
All Bugs	0.0106 (0)	0.0006 (0)	0.0002 (0)

significant advantage over the best performing statistical formulas in each family; despite the fact that, in absolute terms, `Kulczynski2` outperforms the best formula in each family.

> *`Kulczynski2` has no statistically significant advantage over the best formula in other SBFL families (i.e., `Ochiai`, `Naish1` and `GP13`).*

5.3 RQ3: Comparing Statistical Debugging and Dynamic Slicing

How effective is the most effective statistical formula in comparison to dynamic slicing? We compare the performance of the most effective statistical formula (`Kulczynski2`) to that of dynamic slicing (*cf.* Figs. 7 and 8).

We find that, on average, *dynamic slicing is more effective than statistical debugging at fault localization. Slicing is about eight percentage points more effective than the best performing statistical formula* for all programs in our evaluation (*cf.* Fig. 8, Tables 3 *and* 5). For all errors in our study, a programmer using dynamic slicing needs to examine about three-quarters (78%) of those statements that she would need to examine if she used statistical debugging.[20] This result is independent of the type of errors or program. Figure 8 shows that dynamic slicing consistently outperforms statistical debugging for each benchmark, with slicing consistently localizing all faults ahead of statistical debugging.

> *Overall, dynamic slicing was eight percentage points more effective than the best performing statistical debugging formula, i.e.* `Kulczynski2`.

For two-third of bugs (66%, 243 out of 369 errors), dynamic slicing will find the fault earlier than the best performing statistical debugging formula. Figure 7 shows a direct comparison of the scores computed for slicing and statistical debugging. Each 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 diagonal line are better localized using dynamic slicing. For all benchmarks, the majority of the points are above the diagonal

[20] Percentage improvement is measured as $\frac{1-0.794}{1-0.737}$. Note that *score* by itself gives the number of statements that need *not* be examined.

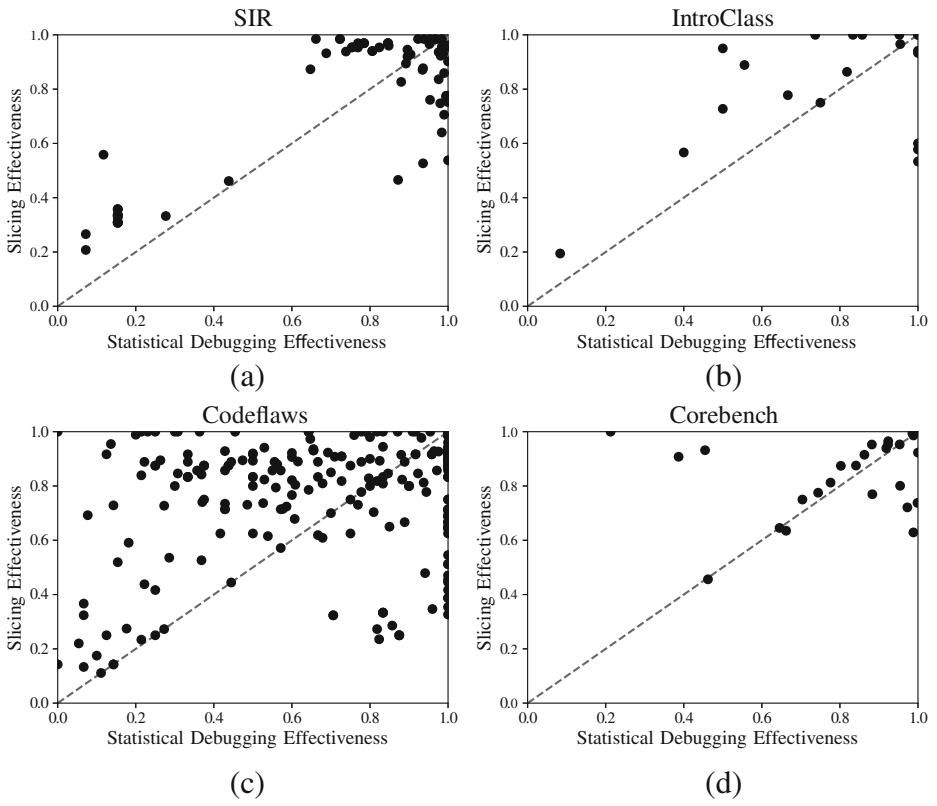

Fig. 7 Direct comparison of fault localization effectiveness between statistical debugging (Kulczynski2) and dynamic slicing (on single faults) in each benchmark

line which indicates that slicing outperforms statistical debugging in most cases. We can see that dynamic slicing consistently outperforms statistical debugging across all benchmarks.

For two-third (66%) of bugs, dynamic slicing locates the fault earlier than the best performing statistical debugging formula, i.e. `Kulczynski2`.

To compare the significance of dynamic slicing and statistical debugging, we compute the odds ratio and conduct a Mann-Whitney *U-test* (*cf. Slicing vs. Kulczynski2 in* Table 7). The odds ratio is in favor of dynamic slicing ($\psi > 1$) for all projects. In particular, slicing is 62% more likely to find a faulty statement earlier than statistical debugging, this likelihood is also statistically significant according to the Mann-Whitney test. The *statistically significant odds ratio* is explained by slicing being more effective than statistical debugging in most cases. For instance, slicing is more effective than statistical debugging for 50 out of 74 bugs in the `SIR` benchmark and for 18 out of 27 bugs in CoREBench.

Dynamic slicing is significantly more likely to find a faulty statement earlier than statistical debugging.

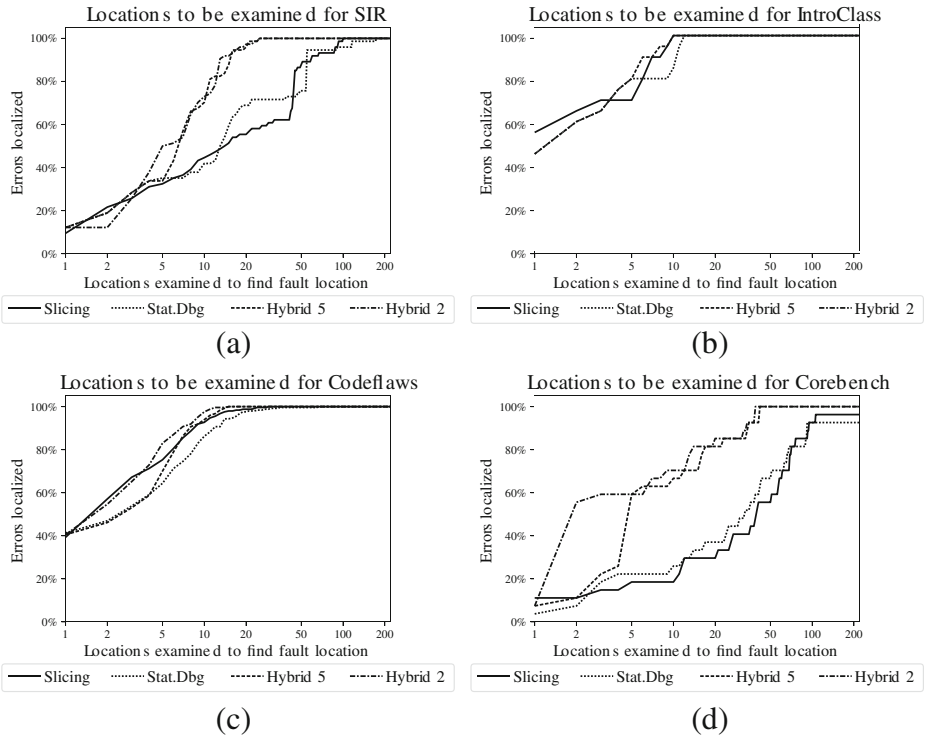

Fig. 8 Cumulative frequency of the locations to be examined, for dynamic slicing vs. statistical debugging vs. the hybrid approach (on single faults) in each benchmark

5.4 RQ4: Sensitiveness of the Hybrid Approach

How many suspicious statements (reported by statistical debugging, i.e. Kulczynski2) should a tool or developer inspect before switching to slicing? We examine the sensitiveness of the hybrid approach to varying absolute values of N. We evaluate how the number of suspicious

Table 7 Statistical Tests for all three Fault Localization Techniques: Odds ratio ψ (Mann-Whitney *U-test* p-values (U) are in brackets), odds ratio with statistically significant p-values determined by Mann-Whitney (*U-test*) are in **bold**

Benchmark	Odds ratio ψ (Mann whitney test score)		
	Slicing vs. `Kulczynski2`	Slicing vs. Hybrid-2	Hybrid-2 vs. `Kulczynski2`
`SIR`	**4.25** (0.0000)	0.81 (0.2568)	**5.44** (0.000)
`IntroClass`	2.16 (0.1087)	0.68 (0.2713)	0.68 (0.2713)
`Codeflaws`	1.16 (0.2094)	**0.41** (0.000)	1.05 (0.3938)
`CoREBench`	2.06 (0.0904)	**0.06** (0.0000)	**16.74** (0.0000)
All Bugs	**1.62** (0.0006)	**0.42** (0.000)	**1.69** (0.0002)

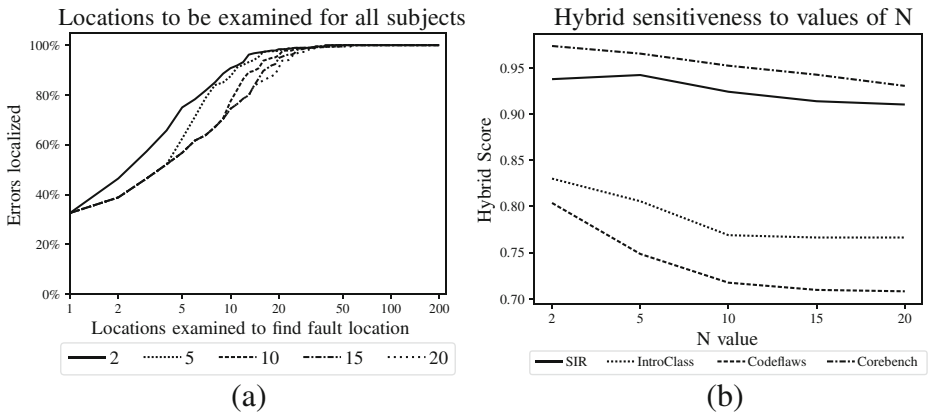

Fig. 9 Hybrid sensitiveness to different values of $N \in \{2, 5, 10, 15, 20\}$ showing **a** the cumulative frequency of locations to be examined for all errors (left), and **b** the effectiveness score for each benchmark using the hybrid approach (right)

statements inspected before switching to slicing influences the effectiveness of the hybrid approach. In particular, we investigated the effect of N values (2, 5, 10, 15, 20) on the performance of the hybrid approach, in order to determine the optimal number of suspicious statements to inspect before switching to slicing.

A programmer that switches to slicing after investigating the top five most suspicious statements can localize more errors than if switching after investigating more suspicious statements. Figure 9 shows the impact of other values of N on the effectiveness of the hybrid approach. Note that the hybrid approach degenerates to dynamic slicing when $N = 0$ and to statistical debugging when N is large (e.g., program size). We see that Kochhar's suggestion of $N = 5$ is a good value for our subjects, in particular inspecting at most five statements before switching to slicing outperforms both slicing and statistical debugging. As we see in Fig. 10, the hybrid approach with $N = 2$ and $N = 5$ outperforms both slicing and statistical debugging (`Kulczynski2`). Hence, *a developer is most effective if she inspects at most five most suspicious statements reported by statistical debugging before switching to slicing.*

A tool using our hybrid approach is most effective when inspecting only the top two most suspicious statements ($N = 2$) reported by statistical debugging, before switching to slicing. Hence, we recommend the use of the hybrid approach (with $N = 2$) for fault localization, and at most five suspicious statements should be inspected before switching to slicing.[21]

The hybrid approach is most effective when a programmer inspects at most two statements ($N = 2$) before switching to slicing.

5.5 RQ5: Effectiveness of the Hybrid Approach

Which technique is the most effective in fault localization? Which technique is more likely to find fault locations earlier? We now investigate the effectiveness of the hybrid approach, in

[21] Further evaluations of the *hybrid approach* use the best values of N (i.e. $N = 2$ and $N = 5$).

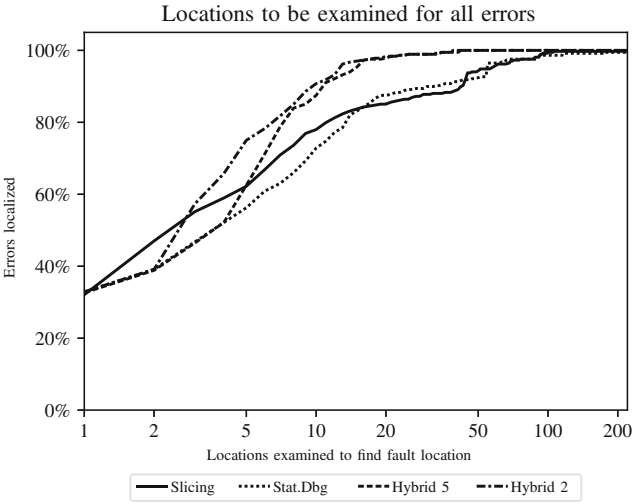

Fig. 10 Cumulative frequency of the locations to be examined for the hybrid approach, in comparison to statistical debugging (Kulczynski2) and dynamic slicing, for all (Single) Faults. As expected, inspecting only the top two suspicious code locations, Hybrid-2 and statistical debugging perform similarly (localizing about 39% of errors each); they are outperformed by dynamic slicing (47% of errors localized). However, inspecting only the top five locations, Hybrid-2 clearly outperforms statistical debugging and dynamic slicing by localizing 75% of errors, while slicing performs better than statistical debugging (62% vs. 56% of errors)

comparison to slicing and statistical debugging. First, we examine the number of program statements that need to be inspected to localize all faults for each technique (Fig. 10), as well as the absolute effectiveness score of each technique (Tables 3, 5 and 8). Then, we evaluate the likelihood of each technique to find the fault locations earlier than the other two techniques (Table 7).

Notably, if the programmer is willing to inspect no more than 20 statements, the hybrid approach will localize the fault location for almost all (98%) of the bugs (*cf.* Table 8 and Fig. 10). In contrast, both statistical debugging and slicing can only localize almost all (98%) faults after inspecting about five times as many statements, i.e. 100 LoC. In fact,

Table 8 Effectiveness of the Hybrid approach with $N = 2$ (i.e., Hybrid-2)

Benchmark	Score	% Errors localized if developer inspects **N** most suspicious LoC			
		5	10	20	30
IntroClass	0.83	90.00	100	100	100
Codeflaws	0.80	83.00	97.57	100	100
CoREBench	0.97	59.26	70.37	85.19	85.19
Real	0.83	81.29	95.24	98.64	98.64
Artificial (SIR)	0.94	50.00	72.97	97.30	100
Avg. (Bugs)	0.844	75.00	90.76	98.37	98.91

if the programmer inspects 20 LoC, slicing and statistical debugging would find the fault location for about 85% and 88% of the bugs, respectively.

The hybrid approach can localize the fault location for almost all (98%) of the bugs after inspecting no more than 20 LoC.

In absolute numbers, the hybrid approach is the most effective fault localization technique, followed by slicing, which is more effective than statistical debugging (see Table 8, Figs. 8 and 10). The hybrid approach ($N = 2$) is about seven percentage points more effective than slicing, and about fifteen percentage points more effective than statistical debugging (*cf.* Tables 3, 5 and 8). Overall, it improves the performance of both slicing and statistical debugging. For instance, a programmer using the hybrid approach needs to examine about half (58%) and three-quarter (75%) of those statements that she would need to examine if she used slicing and statistical debugging, respectively.

The hybrid approach is significantly more effective than slicing and statistical debugging, respectively.

We compute the odds ratio and conduct a Mann-Whitney *U-test*, in order to determine the significance of the hybrid approach. The odds ratio for all projects is strictly in favor of the hybrid approach ($\psi > 1$ in Table 7). Specifically, the hybrid approach is (69%) more likely to find a faulty statement earlier than statistical debugging (*cf. "Hybrid-2 vs. Kulczynski2" in* Table 7*)*. Moreover, a programmer is (42%) less likely to find the fault location early if she localizes with dynamic slicing instead of the hybrid approach (*cf. "Slicing vs. Hybrid-2" in* Table 7*)*.

The *statistically significant odds ratio* is explained by the hybrid approach being more effective than slicing and statistical debugging in most cases. *The majority of bugs is best localized by the hybrid approach.* For more than half of the bugs (56%, 208 out of 369 errors), the hybrid approach will find the fault earlier than both slicing and statistical debugging. In particular, for CoREBench, the hybrid approach is more effective than both techniques for 19 out of 27 bugs, as well as for 33 out of 74 bugs in `SIR`.

The hybrid approach is significantly more likely to find a faulty statement earlier than dynamic slicing and statistical debugging.

5.6 RQ6: Real Errors vs. Artificial Errors

In this section, we evaluate the effect of error type on the effectiveness of an AFL technique, in particular, the difference between evaluating AFL techniques on artificial errors (i.e., `SIR`[22]) versus real errors (i.e., `IntroClass`, `Codeflaws` and `CoREBench`). We examine the performance of each technique on each error type and portray the bias and differences in such evaluations. For real faults, we also evaluated the effect of program

[22]The `SIR` benchmark is the *most used subject for the evaluation of AFL techniques*, especially statistical fault localization (Wong et al. 2016).

size on the effectiveness of an AFL technique. In particular, the difference in the performance of these AFL techniques on *small programs* containing 16 to 18 executable LoC (i.e. `IntroClass` and `Codeflaws`) and *large programs* with about 540 LoC (`CoREBench`), on average (*see* Table 9). Figures 7 and 8 highlight the difference between evaluating a fault localization technique on real or artificial errors. Table 9 summarizes the difference in the effectiveness of each AFL technique when using real or artificial faults, as well as their performance on small or large programs. Tables 3, 5 and 8 also quantify the difference in the effectiveness of all three AFL techniques (i.e., dynamic slicing, statistical debugging and the hybrid approach, respectively) on real and artificial faults.

What is the most effective statistical debugging formula for artificial or real faults? In our evaluation, the error type influences the effectiveness of a statistical debugging formula. `PattSim_2` is the most effective statistical formula for artificial faults (*score=0.85*), this is closely followed by `Ochiai` and `Naish_1` with $score = 0.83$ (*see* Table 4). Meanwhile, for real faults, the most effective statistical formulas are `Kulczynski2` and `Tarantula` with scores 0.76, 0.70 and 0.79 for `IntroClass`, `Codeflaws` and `CoREBench`, respectively (*see* Table 4). Notably, the most effective formula for artificial faults is not the most effective formula for real faults. This implies that the error type can influence the performance of an AFL technique. Thus, we recommended to always evaluate debugging aids using real faults.

The performance of a statistical debugging formula depends on the error type: the most effective formula differs for artificial (`PattSim_2`) and real faults (`Kulczynski2` and `Tarantula`).

How does the effectiveness of statistical debugging compare to that of dynamic slicing, for artificial and real faults? On one hand, dynamic slicing performs worse than statistical debugging on artificial faults (*SIR*): A developer (or tool) has to inspect 21% of the program to find the fault, in contrast to 18% for *Kulczynski2*, on average (*see* Table 9). On the other hand, dynamic slicing performs better than statistical debugging on real faults (i.e., `IntroClass`, `Codeflaws` and `CoREBench`). For real errors, a developer has to inspect (7%) less statements when using dynamic slicing (18%) compared to slicing (25%). Again,

Table 9 Effectiveness of AFL techniques on Real versus Artificial faults, as well as their effectiveness on small and large programs containing real faults (N/A = Not applicable)

Benchmark	Program size (Avg. # LoC)	Percentage (# LoC) of statements Inspected before locating fault		
		Slicing	`Kulczynski2`	Hybrid
`IntroClass`	small (15.5)	17% (3)	24% (4)	17% (3)
`Codeflaws`	small (17.7)	23% (4)	31% (5)	20% (3)
`CoREBench`	large (540.4)	15% (80)	21% (112)	3% (14)
Real	N/A (200.3)	18% (29)	25% (40)	13% (7)
Artificial (`SIR`)	N/A (148.1)	21% (31)	18% (26)	6% (9)
All Bugs	N/A (193.5)	21% (40)	26% (51)	15% (30)

this shows that the error type has a significant influence on the effectiveness of an AFL technique.

Statistical debugging performs better on artificial faults, while dynamic slicing performs better on real faults.

What is the most effective AFL approach on artificial and real faults? The hybrid approach is the most effective AFL approach, outperforming both dynamic slicing and statistical debugging (*see* Table 9). In particular, depending on the error type, a developer or tool using the hybrid approach inspects one-third to less than three-quarter (0.3 to 0.7) of the statements inspected when using dynamic slicing or statistical debugging. This shows that the effectiveness of the hybrid approach is independent of error type.

The hybrid approach is the most effective approach, regardless of error type, i.e. artificial or real faults.

We observed that fault localization effectiveness on artificial errors does not predict results on real faults. In our evaluation, the performance of dynamic slicing and statistical debugging are different depending on the error type. For instance, Table 9 clearly shows that dynamic slicing performs better on real faults, while statistical debugging performs better on artificial faults. This result illustrates that the performance of an AFL technique on artificial faults is not predictive of its performance in practice. Hence, it is pertinent to evaluate AFL techniques on real faults rather than artificial faults, this is in line with the findings of previous studies (Pearson et al. 2017).

The effectiveness of an AFL technique on artificial faults does not predict its effectiveness on real faults.

Does program size affect the effectiveness of individual statistical debugging formulas? Does the most effective SBFL formula vary as program size varies, i.e. small versus large programs? Among real faults, the most effective SBFL formula depends on the program size. Evidently, the most effective formula for small programs is not the most effective formula for large programs: For instance, `Tarantula` and `Kulczynski2` performed significantly better than `DStar` on small programs, but `DStar` was the most effective formula for large programs.[23] Generally, the effectiveness of individual SBFL formulas varies as program size varies. Even though some SBFL formulas performed consistently well across program sizes (e.g. `Tarantula` and `Kulczynski2`), others are specialized for specific

[23] Table 4 shows that `Tarantula` and `Kulczynski2` performed best on small programs with effectiveness scores 0.76 and 0.70 for `IntroClass` and `Codeflaws`, respectively. Despite the fact that `DStar` performs best on large programs (i.e. CoREBench) by slightly outperforming `Tarantula` and `Kulczynski2` (0.80 vs. 0.79); it performed significantly worse than `Tarantula` and `Kulczynski2` on small programs, with effectiveness score 0.62 and 0.56 for `IntroClass` and `Codeflaws`, respectively.

program sizes, e.g. DStar performs better on large programs (CoREBench). These results suggest that program size influences the effectiveness of individual statistical debugging techniques.

*The effectiveness of individual SBFL formulas varies as program sizes varies: **Tarantula** and **Kulczynski2** are the most effective formulas for small programs, but **DStar** is the most effective formula for large programs.*

Does the comparative effectiveness of our AFL techniques (i.e., Hybrid vs. Slicing vs. SBFL) vary as program sizes varies? For real faults, we investigated if there is a difference in the comparative effectiveness of our AFL techniques on small or large programs. Among real faults, the most effective technique is the same across program sizes (*see* Fig. 7 and Table 9): Consistently, the hybrid approach performs best and slicing outperforms statistical debugging, regardless of program size. This observation holds across program sizes, for all three AFL techniques (*see* Fig. 8). These results suggest that program size does not influence the comparative effectiveness of these techniques. In fact, the comparative effectiveness of these AFL techniques is predictable across program sizes; the hybrid approach performs best, followed by slicing then statistical debugging (i.e., Kulczynski2).

For real faults, the comparative effectiveness of our AFL techniques is predictable across program sizes; the hybrid approach performs best, followed by slicing then statistical debugging.

5.7 RQ7: Single Fault vs. Multiple Faults

In this section, we compare the effectiveness of all three AFL techniques on programs with *multiple faults*. Then, we examine the effect of multiple faults on the performance of each technique and the difference between evaluating an AFL technique on single or multiple fault(s). In this experiment, we employ the original single-fault versions of the SIR and IntroClass benchmarks, as well as the multiple-fault versions of the same benchmarks, called SIR-MULT and IntroClass-MULT, respectively. Table 10 highlights the results for single and multiple fault(s) for all AFL techniques, including statistical debugging, hybrid and dynamic slicing. Figures 11, 12 and 13 illustrate the difference in the performance of each technique when given programs with a single fault or multiple faults.

What is the most effective statistical debugging formula for multiple faults? In our evaluation, the most effective SBFL formula for multiple faults is Tarantula (0.8269), from the *popular* SBFL family. It outperforms the other SBFL formulas (*cf.* Table 10 and Fig. 12). For the other statistical debugging families, the most effective formula for multiple faults are DStar (D^2 and D^3), GP02 and m9185 for the *popular*, *human-generated* and *genetically evolved* families, respectively (*cf.* Table 10). The performance of Tarantula is closely followed by that of the single-bug optimal formulas m9185 (0.8249). However, the difference in the performance of m9185 and Tarantula is not statistically significant, i.e. $\psi < 1$ (*odds ratio* $\psi = 0.14$, *Mann-Whitney U-test p-value* $U = 0$). Notably, the most

Table 10 Effectiveness of all AFL techniques on Single and Multiple Faults for `SIR` and `IntroClass` benchmarks

AFL technique	Formula/ approach	`SIR` MULT (*Single*)	`IntroClass` MULT (*Single*)	Average (mean)	
				Programs	Bugs
Popular SBFL	`Tarantula`	**0.8214** (*0.6907*)	**0.8324** (*0.7464*)	**0.8269** (*0.7186*)	**0.7878** (*0.6266*)
	`Ochiai`	0.7796 (*0.7337*)	0.7747 (*0.7464*)	0.7772 (*0.7401*)	0.7560 (*0.6514*)
	`Jaccard`	0.7720 (*0.7029*)	0.7747 (*0.7464*)	0.7734 (*0.7247*)	0.7532 (*0.6342*)
Human Generated SBFL	`Naish_1`	0.7215 (*0.7399*)	0.7638 (*0.7448*)	0.7426 (*0.7423*)	0.7136 (*0.6682*)
	`Naish_2`	0.7484 (*0.7207*)	**0.7747** (*0.7464*)	0.7615 (*0.7336*)	0.7404 (*0.6596*)
	`Russel_Rao`	0.7350 (*0.6088*)	0.7586 (*0.6394*)	0.7468 (*0.6241*)	0.7185 (*0.6051*)
	`Binary`	0.7125 (*0.6283*)	0.7553 (*0.6378*)	0.7339 (*0.633*)	0.6975 (*0.6129*)
	`Wong_1`	0.7350 (*0.6088*)	0.7586 (*0.6394*)	0.7468 (*0.6241*)	0.7185 (*0.6051*)
	D^2	**0.7718** (*0.7345*)	**0.7747** (*0.7464*)	**0.7732** (*0.7404*)	**0.7553** (*0.6523*)
	D^3	0.7680 (*0.7484*)	**0.7747** (*0.7464*)	0.7714 (*0.7474*)	**0.7553** (*0.6611*)
GP Evolved SBFL	`GP_02`	**0.7633** (*0.6725*)	**0.7747** (*0.7157*)	**0.7690** (*0.6941*)	**0.7498** (*0.6133*)
	`GP_03`	0.7473 (*0.6813*)	**0.7747** (*0.6169*)	0.7610 (*0.6491*)	0.7402 (*0.6285*)
	`GP_13`	0.7456 (*0.7211*)	**0.7747** (*0.7464*)	0.7602 (*0.7338*)	0.7398 (*0.6606*)
	`GP_19`	0.7629 (*0.4673*)	0.7558 (*0.6237*)	0.7593 (*0.5455*)	0.7279 (*0.4876*)
Single Bug Optimal SBFL	`PattSim_2`	0.7608 (*0.7537*)	0.7747 (*0.6544*)	0.7677 (*0.704*)	0.7301 (*0.6506*)
	`lex_Ochiai`	0.7532 (*0.7356*)	0.7747 (*0.7464*)	0.7640 (*0.741*)	0.7396 (*0.6646*)
	`m9185`	**0.8183** (*0.7570*)	**0.8315** (*0.7225*)	**0.8249** (*0.7397*)	**0.7664** (*0.6646*)
	`Kulczynski2`	0.7885 (*0.7572*)	0.7747 (*0.7464*)	0.7816 (*0.7518*)	0.7588 (*0.6689*)
Program Slicing	`Dynamic Slicing`	0.8357 (*0.7935*)	0.5487 (*0.8602*)	0.6922 (*0.8269*)	0.7840 (*0.7535*)

Table 10 (continued)

AFL technique	Formula/ approach	SIR MULT (*Single*)	IntroClass MULT (*Single*)	Average (Mean) Programs	Bugs
Hybrid Approach	Hybrid-2	**0.9627** (*0.9358*)	**0.9057** (*0.888*)	**0.9342** (*0.9119*)	**0.9457** (*0.9237*)
	Hybrid-5	0.9505 (*0.9397*)	0.8406 (*0.814*)	0.8955 (*0.8768*)	0.9206 (*0.8974*)

Single Fault Scores are in italics and bracketed, i.e. (*Single*), while Multiple Fault Scores are in normal text. For multiple faults, the best scores for each (sub)category are in **bold**; higher scores are better. For instance, Tarantula is the best performing (popular) statistical debugging formula for *all programs with multiple faults with score 0.8269*, on average

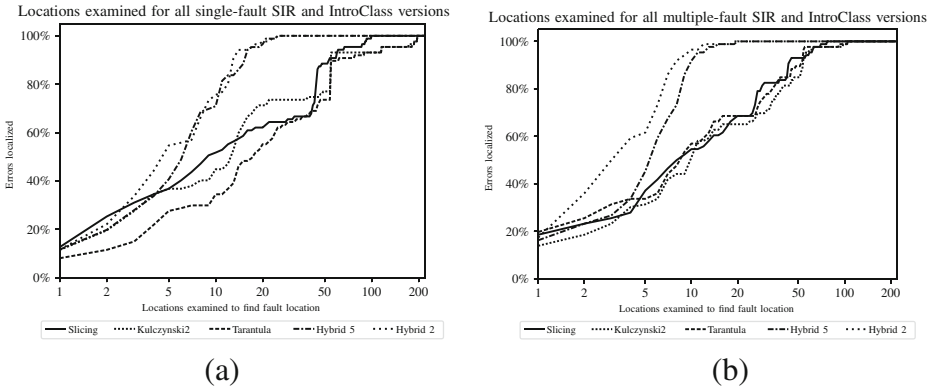

Fig. 11 Cumulative frequency of the locations to be examined for **a** single-fault and **b** multiple-fault versions of the SIR and IntroClass, using the hybrid approach, statistical debugging (Kulczynski2 and Tarantula) and dynamic slicing

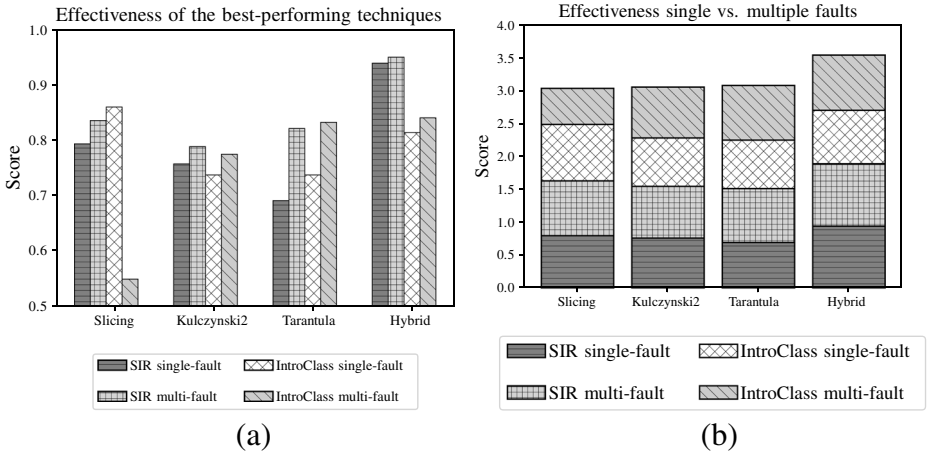

Fig. 12 Effectiveness of each technique for Single and Multiple Fault(s) in SIR and IntroClass: **a** Scores for each benchmark and **b** Scores for both benchmarks

effective single-bug optimal formulas (i.e. `m9185`) outperformed the human-generated and genetically evolved formulas (*cf.* Table 10). This illustrates that *single bug optimal* formulas are also effective for multiple faults, despite being specialized for single faults.

Tarantula is the most effective statistical debugging formula for multiple faults; it outperforms all other statistical debugging formulas.

For multiple faults, how does the effectiveness of statistical debugging (Tarantula) compare to that of dynamic slicing and hybrid? `Tarantula` performs better than dynamic slicing (0.8269 vs. 0.6922) for multiple faults, our results show that the effectiveness of slicing is 16% worse than that of `Tarantula` on multiple faults in the `SIR-MULT` and `IntroClass-MULT` programs (*see* Table 10). This is despite the fact that dynamic slicing (0.8269) outperforms `Tarantula` (0.7186) by 15% on single fault programs (i.e., in `SIR` and `IntroClass` benchmarks). Indeed, there is a 13% decrease in the performance of dynamic slicing on multiple faults. This is evident in Fig. 12a where the performance of dynamic slicing drops for multiple faults for `IntroClass-MULT`. This shows that it is beneficial for an AFL technique to employ coverage data from (numerous) failing test cases when diagnosing programs with multiple faults. As expected, it is more difficult for dynamic slicing to diagnose multiple faults: Since a dynamic slice is constructed for only a single failing test case, it is difficult to account for the effect of multiple faults. Overall, the performance of the hybrid approach remains superior to that of dynamic slicing and statistical debugging, regardless of the number of faults present in the program (*cf.* Table 10, Figs. 11 *and* 12).

Statistical debugging performs better on multiple faults: Tarantula is 19% more effective than dynamic slicing on multiple faults.

Given single or multiple faults, does the effectiveness of an AFL technique improve or worsen? Figure 12a illustrates the difference in the performance of all techniques for single and multiple faults. Results show that all techniques (except dynamic slicing) perform better on multiple faults in comparison to single faults, improvements range from two to 11 percentage points. Notably, `Tarantula`'s performance improved by 11% on multiple faults. Meanwhile, other approaches improved by two to three percentage points, in particular, the hybrid approach, `Kulczynski2` and `DStar` (D^2 and D^3). This illustrates that most AFL approaches—especially SBFL—perform better on multiple faults than single faults (*see* Figs. 12a and 13).

Statistical debugging is better suited for diagnosing multiple faults, while dynamic slicing is more effective at localizing single faults.

Generally, we found that the performance of a technique on programs containing single faults does not predict its performance on multiple faults. For instance, although `Kulczynski2` outperformed the other statistical formulas for single faults (0.7518), it is

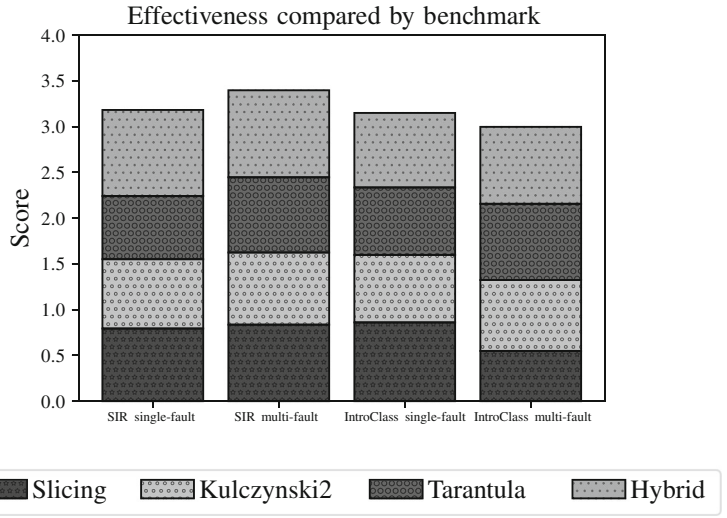

Fig. 13 Effectiveness of each technique on Single and Multiple Faults compared by benchmark, i.e., `SIR` and `IntroClass`

outperformed by `m9185` for multiple faults (0.7816 vs. 0.8249) (*cf.* Table 10). This result is also evident from Fig. 12a and Table 10, where dynamic slicing outperforms statistical debugging (`Kulczynski2`) for `SIR` single faults (0.8269 vs. 0.7518), but statistical debugging (*m9185* and `Kulczynski2`) clearly outperform dynamic slicing for multiple faults. These results suggest that the number of faults in the program influences the effectiveness of an AFL technique.

The effectiveness of an AFL technique on single faults does not predict its effectiveness on multiple faults.

6 Threats to Validity

We discuss the threats to validity for this fault localization study within the framework of Steimann et al. (2013).

6.1 External Validity

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 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 also 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 contain well-maintained open-source C projects with 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 reproducing our experiments for real errors in projects written in other languages as well.
- *EV.4) Test Suites*. The size of a test suite can influence the performance of an AFL technique. Testing strategies that reduce or increase test suite size such as test reduction or test generation methods (i.e. removing tests or generating new tests) have been shown to improve the performance of some AFL techniques (Yang et al. 2017; Yu et al. 2008). We mitigate the effect of test suite size by employing projects with varying test suite sizes (ranging from tens to tens of thousands of test cases) as provided by our subject programs. In our evaluation, we do not generate additional tests or remove any tests from the test suite provided by the benchmarks, in order to simulate the typical debugging scenario for the software project.
- *EV.5) Missing Statements in Slices*. Although, there is a risk of discarding faulty statements during program slicing, dynamic slicing rarely miss faulty statements during fault localization. Reis et al. (2019) found that dynamic slicing reports the faulty statement in the top-ten most suspicious statement 91% of the time. We further mitigate this risk by first inspecting statements in the dynamic slice before inspecting other executable statements. Thus, dynamic slicing (eventually) finds the faulty statement for all bugs in our evaluation.

6.2 Construct Validity

Construct validity refers to the degree to which a test measures what it claims to be measuring. The most immediate threats to construct validity are the following:

- *CV.1) Measure of Effectiveness*. Conforming to the standard (Wong et al. 2016), 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" (Parnin and Orso 2011). However, Steimann et al. (2013) 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 18 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 implementations and experimental results available online for public scrutiny.

7 Future Work

Fault localization based on dependencies still has much room for improvement. For instance:

Cognitive load. In our investigation, we did not consider or model the *cognitive load* it takes to understand the role of individual statements in context. Since following dependencies in a program is much more likely to stay within same or similar contexts than statistical debugging, where the ranked suspicious lines can be strewn arbitrarily over the code, we would expect dependency-based techniques to take a lead here. The seminal study of Parnin and Orso (2011) found that ranked lists of statements are hardly helping human programmers—let us find out which techniques work best for humans.

Alternate search techniques. Furthermore, there would be other search strategies along dependencies (for instance, starting with the input, and progressing forward through a program; starting at some suspicious or recently changed location; or moving along coarse-grained functions first, and fine-grained lines later) that may be even more efficient both in terms of nodes visited as well as from the assumed cognitive load. Again, all this calls for more human studies in debugging.

Experimental techniques. such as delta debugging (Zeller and Hildebrandt 2002) offer another means to reduce the cognitive load—by systematically narrowing down the conditions under which a failure occurs. The work of Burger and Zeller (2011) on minimization of calling sequences with delta debugging showed dramatic improvements over dynamic slicing, reducing "the search space to 13.7% of the dynamic slice or 0.22% of the source code". In a recent human study, delta debugging "statistically significantly increased programmers efficiency in failure detection, fault localization and fault correction." (Hammoudi et al. 2015).

Symbolic techniques. Finally, following dependencies is still one of the simplest methods to exploit program semantics. Applying symbolic execution and constraint solving would narrow down the set of possible faults. Model-based debugging (Wotawa et al. 2002) was one of the first to apply this idea in practice; the more recent BUGASSIST work of Jose and Majumdar "quickly and precisely isolates a few lines of code whose change eliminates the error." (Jose and Majumdar 2011).

All these techniques would profit from wider evaluation and assessment; however, they can also be joined and combined; for instance, one could start with suspicious statements as indicated by statistical fault localization, follow dependencies from there, and skip influences deemed impossible by symbolic analysis. What we need, though, is true defects which we can use to compare the techniques with—and a willingness to actually compare state of the art techniques, as we do in this paper.

8 Related Work

8.1 Evaluation of Fault Localization Techniques

The effectiveness of various fault localization approaches have been studied by several colleagues, see Wong et al. (2016). Most papers investigated the effects of program, test and bug features on the effectiveness of statistical debugging. Abreu et al. (2009a) examined

the effects of the number of passing and failing test cases on the effectiveness of statistical debugging, they established that the suspiciousness scores stabilize starting from an average six (6) failing and twenty (20) passing test cases. Pearson et al. (2017) evaluated the difference between evaluating fault localization techniques on real faults versus artificial faults, using two main techniques, namely statistical debugging and mutation-based fault localization. Notably, their evaluation results shows that results on artificial faults do not predict results on real faults for both techniques, and a hybrid technique is significantly better than both techniques. Keller et al. (2017) and Heiden et al. (2019) evaluated the effectiveness of statistical fault localization on real world large-scale software systems. The authors found that, for realistic large-scale programs, the accuracy of statistical debugging is not suitable for human developers. In fact, the authors emphasize the obvious need to improve statistical debugging with contextual information such as information from the bug report or from version history of the code lines. In contrast to our work, none of these papers evaluated the fault localization effectiveness of program slicing, nor compare the effectiveness of slicing to that of statistical debugging.

A few approaches have evaluated the effectiveness of dynamic slices in fault localization (Zhang et al. 2005, 2007; Li and Orso 2020). In particular, Zhang et al. (2005) evaluated the effectiveness of three variants of dynamic slicing algorithms, namely data slicing, full (dynamic) slicing, and relevant slicing. Recently, Li and Orso (2020) proposed the concept of *potential memory-address dependence* (PMD) to improve the accuracy of dynamic slicing. Traditional dynamic dependence graphs (DDG) do not account for PMDs since they are not actual data or control dependencies in the program (Li and Orso 2020). In particular, PMD-slicer determines the potential memory dependencies in a program and represents them on the DDG. This allows developers to detect faults that are due to program assignments that modify the wrong code location. Like our study, these papers also found that (variants of) program slicing considerably reduces the number of program statements that need to be examined to locate faulty statements. However, in contrast to our study, these papers have not empirically compared the performance of dynamic slicing to that of statistical debugging.

8.2 Improvements of Statistical Fault Localization

Several authors have proposed approaches to improve statistical fault localization. Most approaches are focused on reducing the program spectra (i.e. the code coverage information) fed to statistical debugging, sometimes by using delta debugging (Christi et al. 2018), program slicing (Alves et al. 2011; Lei et al. 2012; Guo et al. 2018), test generation (Liu et al. 2017), test prioritization (Zhang et al. 2017) or machine learning (Zou et al. 2019; Le et al. 2016). In particular, some techniques apply program slicing to reduce the program spectra fed to statistical debugging formulas (Shu et al. 2017; Alves et al. 2011; Liu et al. 2016; Lei et al. 2012; Guo et al. 2018). The popular page rank algorithm has been used to boost statistical debugging effectiveness by estimating the contributions of different tests to re-compute program spectral (Zhang et al. 2017). Machine learning algorithms (such as learning to rank) have also been used to improve the effectiveness of statistical debugging (Le et al. 2016; Zou et al. 2019). Besides, BARINEL employs a combination of bayesian reasoning and and statistical debugging to improve fault localization effectiveness, especially for programs with multiple faults (Abreu et al. 2009b). BARINEL combines statistical debugging and *model-based diagnosis* (MBD), i.e., logic reasoning over a behavioral model to deduce multiple-fault candidates: The goal is to overcome the high computational complexity of typical MBD. Search-based test generation has also been combined with SBFL,

in order to improve the performance of statistical debugging for Simulink models (Liu et al. 2017). However, none of these papers localize faults by following control and data dependencies in the program, i.e. they do not directly use program slicing as a fault localization technique.

Zou et al. (2019) found that the combination of fault localization techniques improves over individual techniques, the authors recommend that future fault localization techniques should be evaluated in the combined setting. Wotawa (2010) proposed a combination of model-based debugging and program slicing for fault localization, called *slicing hitting set computation* (SHSC). In contrast to our work, SHSC combines slices of faulty variables, which causes undesirable high ranking of statements executed in many test cases (Hofer and Wotawa 2012). To address this, Hofer and Wotawa also proposed SENDYS—a combination of statistical debugging and SHSC to improve the ranking of faulty statements (Hofer and Wotawa 2012). The focus of this work is to provide fault locations at a finer granularity than program blocks. In contrast to dynamic slicing, SENDYS analyzes the execution information from both passing and failing test cases and uses statistical debugging results as a-priori fault probabilities of single statements in SHSC (Hofer and Wotawa 2012).

9 Conclusion and Consequences

As it comes to debugging, *dynamic slicing remains 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 recommend that following dependencies should remain the primary method of fault localization—it is a safe and robust technique that will get programmers towards the goal.

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 dependencies just as human debuggers.

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 (Zeller and Hildebrandt 2002; Hammoudi et al. 2015), or executions (Burger and Zeller 2011) may dramatically improve fault localization by focusing on relevant parts of the execution. *Grammar-based techniques* that debug inputs (Kirschner et al. 2020), generalize inputs (Gopinath et al. 2020) or determine the circumstances of failure (Kampmann et al. 2020) may provide contextual information for developers during debugging by focusing on input features that explain failures. *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" (Jose and Majumdar 2011). 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 as an artifact, in order to support scrutiny, evaluation, reproduction, and extension: https://tinyurl.com/HybridFaultLocalization

Acknowledgements We thank the anonymous reviewers for their helpful comments. This work was funded by Deutsche Forschungsgemeinschaft, Project "Extracting and Mining of Probabilistic Event Structures from Software Systems (EMPRESS)".

References

Abreu R, Zoeteweij P, van Gemund AJC (2006) An evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim international symposium on dependable computing, PRDC'06, pp 39–46

Abreu R, Zoeteweij P, van Gemund AJC (2007) On the accuracy of spectrum-based fault localization. In: Proceedings of the testing: academic and industrial conference practice and research techniques—MUTATION, TAICPART-MUTATION '07, pp 89–98

Abreu R, Zoeteweij P, Golsteijn R, van Gemund AJC (2009a) A practical evaluation of spectrum-based fault localization. J Syst Softw 82(11):1780–1792

Abreu R, Zoeteweij P, Van Gemund AJ (2009b) Spectrum-based multiple fault localization. In: 2009 IEEE/ACM international conference on automated software engineering. IEEE, pp 88–99

Agrawal H, Horgan JR (1990) Dynamic program slicing. In: Proceedings of the ACM SIGPLAN 1990 conference on programming language design and implementation, PLDI '90, pp 246–256

Agrawal H, DeMillo RA, Spafford EH (1990) Efficient debugging with slicing and backtracking. Tech. rep., Purdue University

Agrawal H, Horgan JR, Krauser EW, London S (1993) Incremental regression testing. In: Proceedings of the conference on software maintenance, ICSM '93, pp 348–357

Alves E, Gligoric M, Jagannath V, d'Amorim M (2011) Fault-localization using dynamic slicing and change impact analysis. In: 2011 26th IEEE/ACM international conference on automated software engineering (ASE 2011). IEEE, pp 520–523

Assiri FY, Bieman JM (2017) Fault localization for automated program repair: effectiveness, performance, repair correctness. Software Quality Journal 25(1):171–199. Springer

Binkley D, Gold N, Harman M (2007) An empirical study of static program slice size. ACM Trans Softw Eng Methodol 16(2)

Böhme M, Roychoudhury A (2014) Corebench: studying complexity of regression errors. In: Proceedings of the 23rd ACM/SIGSOFT international symposium on software testing and analysis, ISSTA, pp 105–115

Böhme M, Soremekun EO, Chattopadhyay S, Ugherughe E, Zeller A (2017) Where is the bug and how is it fixed? An experiment with practitioners. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 117–128

Burger M, Zeller A (2011) Minimizing reproduction of software failures. In: Proceedings of the 2011 international symposium on software testing and analysis, ISSTA '11, pp 221–231

Chen MY, Kiciman E, Fratkin E, Fox A, Brewer E (2002) Pinpoint: problem determination in large, dynamic internet services. In: Proceedings of the 2002 international conference on dependable systems and networks, DSN '02, pp 595–604

Christi A, Olson ML, Alipour MA, Groce A (2018) Reduce before you localize: delta-debugging and spectrum-based fault localization. In: 2018 IEEE international symposium on software reliability engineering workshops. IEEE, ISSREW, pp 184–191

DiGiuseppe N, Jones JA (2011) On the influence of multiple faults on coverage-based fault localization. In: Proceedings of the 2011 international symposium on software testing and analysis, pp 210–220

Gopinath R, Kampmann A, Havrikov N, Soremekun EO, Zeller A (2020) Abstracting failure-inducing inputs. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp 237–248

Grissom R, Kim J (2005) Effect sizes for research: a broad practical approach. Lawrence Erlbaum

Guo A, Mao X, Yang D, Wang S (2018) An empirical study on the effect of dynamic slicing on automated program repair efficiency. In: 2018 IEEE international conference on software maintenance and evolution (ICSME). IEEE, pp 554–558

Gyimóthy T, Beszédes A, Forgács I (1999) 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, ESEC/FSE-7, pp 303–321

Hammoudi M, Burg B, Bae G, Rothermel G (2015) 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. ESEC/FSE 2015. ACM, New York, pp 333–344. https://doi.org/10.1145/2786805.2786846

Heiden S, Grunske L, Kehrer T, Keller F, van Hoorn A, Filieri A, Lo D (2019) An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. Softw: Pract Exp 49(8):1197–1224

Hofer B, Wotawa F (2012) Spectrum enhanced dynamic slicing for better fault localization. In: ECAI, vol 12, pp 420–425

Hutchins M, Foster H, Goradia T, Ostrand T (1994) Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In: Proceedings of 16th international conference on software engineering. IEEE, pp 191–200

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering, ASE '05, pp 273–282

Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of the 24th international conference on software engineering, ICSE '02, pp 467–477

Jose M, Majumdar R (2011) Cause clue clauses: error localization using maximum satisfiability. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation, PLDI '11. ACM, New York, pp 437–446. https://doi.org/10.1145/1993498.1993550

Kampmann A, Havrikov N, Soremekun EO, Zeller A (2020) When does my program do this? Learning circumstances of software behavior. In: Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 1228–1239

Keller F, Grunske L, Heiden S, Filieri A, van Hoorn A, Lo D (2017) A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In: 2017 IEEE international conference on software quality, reliability and security (QRS). IEEE, pp 114–125

Kim D, Nam J, Song J, Kim S (2013) Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 international conference on software engineering, ICSE '13, pp 802–811

Kirschner L, Soremekun E, Zeller A (2020) Debugging inputs. In: 2020 IEEE/ACM 42nd international conference on software engineering (ICSE). IEEE

Kochhar PS, Xia X, Lo D, Li S (2016) Practitioners' expectations on automated fault localization. In: Proceedings of the 25th international symposium on software testing and analysis. ACM, pp 165–176

Korel B, Laski J (1988) Dynamic program slicing. Inf Process Lett 29(3):155–163

Landsberg D (2016) Methods and measures for statistical fault localisation. PhD thesis, University of Oxford

Landsberg D, Chockler H, Kroening D, Lewis M (2015) Evaluation of measures for statistical fault localisation and an optimising scheme. In: International conference on fundamental approaches to software engineering. Springer, pp 115–129

Le TDB, Lo D, Le Goues C, Grunske L (2016) A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th international symposium on software testing and analysis, pp 177–188

Le Goues C, Nguyen T, Forrest S, Weimer W (2012) Genprog: a generic method for automatic software repair. IEEE Trans Softw Eng 38(1):54–72

Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The manybugs and introclass benchmarks for automated repair of c programs. IEEE Trans Softw Eng 41(12):1236–1256

Lei Y, Mao X, Dai Z, Wang C (2012) Effective statistical fault localization using program slices. In: 2012 IEEE 36th annual computer software and applications conference. IEEE, pp 1–10

Li X, Orso A (2020) More accurate dynamic slicing for better supporting software debugging. In: 2020 IEEE 13th international conference on software testing, validation and verification (ICST). IEEE, pp 28–38

Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI (2005) Scalable statistical bug isolation. In: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation, PLDI '05, pp 15–26

Lin Y, Sun J, Tran L, Bai G, Wang H, Dong J (2018) Break the dead end of dynamic slicing: localizing data and control omission bug. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 509–519

Liu B, Lucia NejatiS, Briand LC, Bruckmann T (2016) Simulink fault localization: an iterative statistical debugging approach. Softw Test Verif Reliab 26(6):431–459

Liu B, Nejati S, Briand LC et al (2017) Improving fault localization for simulink models using search-based testing and prediction models. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 359–370

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. Ann Math Stat 50–60

Naish L, Lee HJ (2013) Duals in spectral fault localization. In: 2013 22nd Australian software engineering conference. IEEE, pp 51–59

Naish L, Lee HJ, Ramamohanarao K (2011) A model for spectra-based software diagnosis. ACM Trans Softw Eng Methodol (TOSEM) 20(3):11
Nguyen HDT, Qi D, Roychoudhury A, Chandra S (2013) Semfix: program repair via semantic analysis. In: Proceedings of the 2013 international conference on software engineering, ICSE '13, pp 772–781
Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 international symposium on software testing and analysis, ISSTA '11. ACM, New York, pp 199–209. https://doi.org/10.1145/2001420.2001445
Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B (2017) Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International conference on software engineering (ICSE). IEEE, pp 609–620
Perez A, Abreu R, d'Amorim M (2017) Prevalence of single-fault fixes and its impact on fault localization. In: 2017 IEEE international conference on software testing, verification and validation (ICST). IEEE, pp 12–22
Qi D, Nguyen HDT, Roychoudhury A (2013) Path exploration based on symbolic output. ACM Trans Softw Eng Methodol 22(4):32:1–32:41
Qi Y, Mao X, Lei Y, Dai Z, Wang C (2014) The strength of random search on automated program repair. In: Proceedings of the 36th international conference on software engineering, ICSE 2014, pp 254–265
Reis S, Abreu R, d'Amorim M (2019) Demystifying the combination of dynamic slicing and spectrum-based fault localization. In: IJCAI, pp 4760–4766
Renieris M, Reiss SP (2003) Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE international conference on automated software engineering. IEEE, pp 30–39
Russel PF, Rao TR et al (1940) On habitat and association of species of anopheline larvae in south-eastern Madras. Journal of the Malaria Institute of India 3:153–178
Shu T, Wang L, Xia J (2017) Fault localization using a failed execution slice. In: 2017 International conference on software analysis, testing and evolution (SATE). IEEE, pp 37–44
Steimann F, Frenkel M, Abreu R (2013) Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Proceedings of the 2013 international symposium on software testing and analysis, ISSTA 2013, pp 314–324
Tan SH, Yi J, Mechtaev S, Roychoudhury A et al (2017) Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In: 2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C). IEEE, pp 180–182
Tip F (1995) A survey of program slicing techniques. J Program Lang 3(3):121–189
Weiser M (1981) Program slicing. In: Proceedings of the 5th international conference on software engineering, ICSE '81, pp 439–449
Weiser M (1982) Programmers use slices when debugging. Commun ACM 25(7):446–452
Wong WE, Qi Y, Zhao L, Cai KY (2007) Effective fault localization using code coverage. In: Computer software and applications conference, 2007. COMPSAC 2007. 31st annual international, vol 1. IEEE, pp 449–456
Wong WE, Debroy V, Li Y, Gao R (2012) Software fault localization using dstar (d*). In: 2012 IEEE sixth international conference on software security and reliability. IEEE, pp 21–30
Wong WE, Debroy V, Gao R, Li Y (2013) The dstar method for effective software fault localization. IEEE Trans Reliab 63(1):290–308
Wong WE, Gao R, Li Y, Abreu R, Wotawa F (2016) A survey on software fault localization. IEEE Trans Softw Eng p preprint
Wotawa F (2010) Fault localization based on dynamic slicing and hitting-set computation. In: 2010 10th international conference on quality software. IEEE, pp 161–170
Wotawa F, Stumptner M, Mayer W (2002) Model-based debugging or how to diagnose programs automatically. In: Proceedings of the 15th international conference on industrial and engineering applications of artificial intelligence and expert systems: developments in applied artificial intelligence, IEA/AIE '02. Springer, London, pp 746–757. http://dl.acm.org/citation.cfm?id=646864.708248
Xie X, Chen TY, Kuo FC, Xu B (2013a) A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans Softw Eng Methodol (TOSEM) 22(4):31
Xie X, Kuo FC, Chen TY, Yoo S, Harman M (2013b) Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In: International symposium on search based software engineering. Springer, pp 224–238
Yang J, Zhikhartsev A, Liu Y, Tan L (2017) Better test cases for better automated program repair. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 831–841
Yoo S (2012) Evolving human competitive spectra-based fault localisation techniques. In: International symposium on search based software engineering. Springer, pp 244–258

Yu Y, Jones J, Harrold MJ (2008) An empirical study of the effects of test-suite reduction on fault localization. In: 2008 ACM/IEEE 30th international conference on software engineering. IEEE, pp 201–210
Zeller A, Hildebrandt R (2002) Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng 28(2):183–200. https://doi.org/10.1109/32.988498
Zhang X, He H, Gupta N, Gupta R (2005) Experimental evaluation of using dynamic slices for fault location. In: Proceedings of the sixth international symposium on automated analysis-driven debugging, pp 33–42
Zhang X, Gupta N, Gupta R (2007) A study of effectiveness of dynamic slicing in locating real faults. Empir Softw Eng 12(2):143–160
Zhang M, Li X, Zhang L, Khurshid S (2017) Boosting spectrum-based fault localization using pagerank. In: Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis, pp 261–272
Zheng AX, Jordan MI, Liblit B, Aiken A (2003) Statistical debugging of sampled programs. In: Advances in neural information processing systems, pp 9–11
Zheng AX, Jordan MI, Liblit B, Naik M, Aiken A (2006) Statistical debugging: simultaneous identification of multiple bugs. In: Proceedings of the 23rd international conference on machine learning, pp 1105–1112
Zou D, Liang J, Xiong Y, Ernst MD, Zhang L (2019) An empirical study of fault localization families and their combinations. IEEE Trans Softw Eng 47(2, Feb. 1 2021):332–347

Ezekiel Soremekun is a Research Associate at the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg. Ezekiel's research is focussed on automated debugging, software testing and program analysis.

Lukas Kirschner works as student research assistant at Saarland University in Saarbrücken, Germany, where he completed his bachelor's degree in computer science. His work focuses on automated debugging and software testing techniques.

Marcel Böhme is an ARC DECRA Fellow and a Senior Lecturer at Monash University in Australia. Marcel's research is focussed on automated vulnerability detection, analysis, testing, debugging, and repair of large software systems, where he investigates the underlying foundations as well as practical techniques. He received his PhD from the National University of Singapore.

Andreas Zeller is faculty at the CISPA Helmholtz Center for Information Security and professor for Software Engineering at Saarland University, both in Saarbrücken, Germany. His research on automated debugging, mining software archives, specification mining, and security testing has proven highly influential. Zeller is an ACM Fellow and holds an ACM SIGSOFT Outstanding Research Award.