Evaluating the Impact of Experimental Assumptions in Automated Fault Localization

Ezekiel Soremekun*†, Lukas Kirschner‡, Marcel Böhme§, and Mike Papadakis†,
*RHUL, UK, †SnT, Luxembourg, ‡Saarland University, Germany, §MPI-SP, Germany; Monash University, Australia
ezekiel.soremekun@uni.lu, kirschlu@gmail.com, marcel.boehme@acm.org, michail.papadakis@uni.lu

***Abstract*—Much research on automated program debugging often assumes that bug fix location(s) indicate the faults' root causes and that root causes of faults lie within single code elements (statements). It is also often assumed that the number of statements a developer would need to inspect before finding the first faulty statement reflects debugging effort. Although intuitive, these three assumptions are typically used (55% of experiments in surveyed publications make at least one of these three assumptions) without any consideration of their effects on the debugger's effectiveness and potential impact on developers in practice. To deal with this issue, we perform controlled experimentation, split testing in particular, using 352 bugs from 46 open-source C programs, 19 Automated Fault Localization (AFL) techniques (18 statistical debugging formulas and dynamic slicing), two (2) state-of-the-art automated program repair (APR) techniques (GenProg and Angelix) and 76 professional developers. Our results show that these assumptions conceal the difficulty of debugging. They make AFL techniques appear to be (up to 38%) more effective, and make APR tools appear to be (2X) less effective. We also find that most developers (83%) consider these assumptions to be unsuitable for debuggers and, perhaps worse, that they may inhibit development productivity. The majority (66%) of developers prefer debugging diagnoses without these assumptions twice as much as with the assumptions. Our findings motivate the need to assess debuggers conservatively, i.e., without these assumptions.**

I. Introduction

Automated fault localization (AFL) techniques suggest locations in the program that explain the *root cause* of an observed program failure [1]. Given a program failure and one or more failing (and passing) test cases, the goal of AFL is to provide a (sorted) list of program locations that explain the cause of the failure. The aim of AFL is thus, to save *developer* effort, time and resources during debugging (assisting the manual fault localization). Fault localization is also a prerequisite for bug fixing, hence it is relevant for *automated program repair* (APR) tools. It allows APR tools to identify likely candidates for fixing. Indeed, several fault localization techniques have been deployed to aid APR methods [2, 3, 4, 5, 1].

Researchers often make experimental assumptions when evaluating the effectiveness of AFL techniques in the *lab*. Table I provides an overview of three common assumptions employed when evaluating AFL effectiveness and shows examples of these assumptions, drawn from the literature. The first one is that *bug fix location(s) (aka "Fix Location") can be used as substitute of the actual fault location(s)* (aka "root cause") [1]. The second is that AFL effectiveness evaluations can be performed by counting the number of statements a developer would need to inspect before finding the *first* faulty statement, even if there are several faulty statements [6, 7, 8, 9]. This is called the *perfect bug understanding* (PBU) assumption. Even though in practice, the developer may need to inspect more statements before finding all faulty statements. The third assumption is that the root causes of bugs resides in a single contiguous fault location, even though in practice bugs may span multiple contiguous locations [1, 10, 9].

We find that these assumptions are prevalent both in debugging literature and frequently used bug datasets (Section III). This means that they can lead to a mismatch between debugging evaluations in the *lab* and *development practice* [11]. Indeed, such assumptions influence the adoption and practicality of debuggers in practice [12, 13]. For these reasons it is pertinent to examine their effect on the perceived effectiveness of debuggers as well as the productivity of developers.

To address this issue, we conduct a large empirical study to evaluate the impact of these three assumptions on the measured effectiveness of AFL techniques and APR tools. Our experience has shown that these assumptions are important and are frequently made by researchers. We thus, systematically validate the prevalence of our observation (Section III). As we find them important, we further conduct a user study to evaluate how these assumptions impact developers in practice. In our experiments, we employed controlled experimentation (*split - A/B testing*) to examine the current debugging evaluation settings (*with* each assumption) versus its absence (i.e., *without* each assumption) [20, 21]. The goal is to understand the impact of these assumptions in practice. Our experiments were performed using four well-known C benchmarks with 35 tools, 46 programs and 352 bugs. Our user study involved 76 professional developers debugging eight buggy programs in 16 debugging settings.

To the best of our knowledge, this is the *first* study to provide empirical evidence on the impact of these assumptions on debuggers and software practice. We are also the *first* to provide evidence that these assumptions are *highly prevalent*. Overall our work makes the following contributions:

- **Prevalence of Assumptions:** We provide empirical evidence that there are three (3) highly prevalent debugging assumptions in research *literature and bug datasets*. We found that majority (55%) of the reported experiments in the literature make at least one of these assumptions (**RQ1** Section III).
- **Measurement of AFL Effectiveness:** We demonstrate that an AFL tool evaluated with these assumptions might ap-

Table I
EXAMPLES OF REAL-WORLD BUGGY PROGRAMS SHOWCASING EACH ASSUMPTION, INCLUDING SAMPLE DIAGNOSES AND QUOTES FROM LITERATURE ILLUSTRATING EACH ASSUMPTION AND THEIR USE IN DEBUGGING LITERATURE AND BUG DATASETS

	Perfect Bug Understanding (PBU)	**Using Fixes as Fault Location**	**Single Faulty Location**
Example Program	Listing 1. checksum 3b2376ab 006	Listing 2. SpamAssassin BID-6679	Listing 3. grade 531924c0 001
Failing Test Case	The program outputs the same checksum for ''zzz'' and ''zzx''	A buffer overflow occurs for strings of a certain length ending with '\n'	Method user_grade returns "A" if the grade is equal to one of the thresholds
Diagnosis *with* assumption	The fault localization points to line **3** as the faulty location, since the if statement is wrong.	Line **6** is faulty: If 4 is subtracted from the limit, the buffer overflow does not occur anymore.	Line **6** is faulty: The > operator needs to be changed to a >= operator.
Diagnosis *without* assumption	Lines **3**, **4** and **5** are chosen as faulty lines by the fault localization, since the if statement needs to be deleted to fix the bug.	Line **3** is faulty: The buffer needs to be allocated large enough to acommodate the whole string.	Lines **6**, **8** and **10** are faulty: All the > operators must be changed to >= operators.
Quotes *with* assumption	"Recall at Top-K: is the number of faults with at least one faulty statement that is correctly predicted in the ranked list of K statements. "[14]	"for release 2.0 of defects4j, we searched among the fault-fix patches included with the release. we found a total of 228 program versions fitting our criteria."[15]	"We focus on the single fault localisation, i.e. we assume that there exists a single faulty statement in the program. "[16]
Quotes *without* Assumption	"This metric is used to measure how many faults can be located within top k program elements among all candidates. "[17]	"these structural errors are often detected through manual code reviews which is time consuming."[18]	"When multiple statements have the same suspicious score, we use the average rank to present their final rankings "[19]

```
1 void add(int *i, char *c){
2   *i = *i + *c;
3   if (*i >= 256){
4     *i = 0;
5   }
6 }
7 int main(){
8 ...
9   while (c != '\n'){
10    add(&i, &c);
11    scan_data(&c);
12  }
13  i = (i % 64) + 32;
14  printf("Check sum is %c\n", i);
15 ...
16 }
```

```
1 void message_write (char *msg, int
      len)
2 {
3   char buffer[BUFSZ];
4   int i;
5   int j;
6   int limit = BUFSZ - 1;
7   for (i=0; i<len; ) {
8     for (j=0; i<len && j<limit; ){
9       if (i + 1 < len
10          && msg[i] == '\n'
11          && msg[i+1]== '.') {
12        buffer[j] = msg[i];
13        j++;
14        i++;
15 ...
16 }
```

```
1 char
2 user_grade(float percent, float a,
      float b, float c, float d)
3 {
4   if (percent < d)
5     return 'F';
6   else if ((percent > d) && (
      percent < c))
7     return 'D';
8   else if ((percent > c) && (
      percent < b))
9     return 'C';
10  else if ((percent > b) && (
      percent < a))
11    return 'B';
12  else
13    return 'A';
14 }
```

pear (38%) more effective than it actually is, consequently concealing the difficulty of debugging (**RQ2** Section V).

- **Proxy Effect on APR Tools:** We provide empirical evidence that these assumptions have a proxy effect on the measured effectiveness of APR methods. An APR tool evaluated using these assumptions might appear half as effective, and (3X) less expensive, than it is (**RQ3** Section V).
- **Developers Productivity:** We also show that professional developers prefer debugging diagnoses to be provided *without* these assumptions. A majority (up to 83%) of developers prefer debugging diagnoses *without* the assumption up to 4X as much as *with* the assumption. Participants also found these assumptions to be *unsound*, *unrealistic* and *severe* in software practice (**RQ4** Section VII).

II. DEFINITIONS AND RESEARCH QUESTIONS

A. Definition of Assumptions

EA1. Perfect Bug Understanding (PBU): This means that AFL techniques are evaluated by computing the number of statements examined until the *first* faulty statement is found [10, 6, 7, 8, 1, 9]. Researchers assume that locating and examining a single faulty program statement (out of several faulty statements) is sufficient to explain and fix the bug [1]. That is, measuring the starting point to initiate the bug-fixing process is sufficient for diagnoses, rather than providing the complete set of code that must be modified to fix the bug [1]. The left most column of Table I shows an example of the diagnoses provided *with* and *without* this assumption. In this example, PBU means the effectiveness of a debugger is measured by locating any first faulty statement (e.g., line three). Meanwhile, debugging *without* the PBU assumption means a debugger must locate all faulty statements (lines 3, 4, and 5). This paper investigates the impact of the PBU assumption by examining the effectiveness of AFL/APR tools and developer's debugging productivity, *with* versus *without* the assumption. This work shows that developers find this assumption to be *unrealistic in practice* (Section VII). Besides, PBU inflates the effectiveness of debuggers (Section V), despite being commonly used in debugging literature (Section III).

EA2. Using Fix Locations as Root Cause Bug Diagnosis (aka "Fix Location"): In the evaluation of AFL techniques, benchmarks with developer-provided fault locations are often unavailable [13]. In the absence of ground truth fault locations, it is common to use fix locations as substitute fault locations [10, 6, 7, 8, 9]. Clearly, these locations are directly related with removing the observed bug, rather than explaining the cause of the failure. The middle column of Table I shows a simple buggy program with a buffer overflow, where the root cause and the fix location are different. In this example, the bug fix resolves the symptom/effect of the bug (in line 6). However, the developer-provided root cause of the bug is the static allocation of buffer size (in line 3) which if fixed (e.g., via dynamic allocation) resolves the bug for all proceeding use of the buffer beyond the fixed statement (line

Table II
EXCERPT OF ANALYZED PUBLICATIONS (2017 TO MID (JUNE) 2022)

Venue	#Papers	Sample Papers
ICSE	(9)	[9, 22, 23, 24, 25]
FSE	(2)	[26, 27]
ASE	(11)	[28, 29, 17, 30, 31]
MSR	(2)	[32, 33]
TOSEM	(6)	[34, 35, 18, 36, 16, 37]
EMSE	(4)	[38, 39, 40, 41]
TSE	(29)	[42, 43, 44, 45]

6). Indeed, for real-world buggy programs, root causes often do not intersect with the fix location, in fact they may be far apart (*see* Section III). Thus, evaluating debuggers in this setting conceals the difficulty of debugging (Section V) and is unrealistic in debugging practice (Section VII).

EA3. Assuming Single Fault Location (aka "Single Fault Location"): Researchers often assume that the diagnoses for a faulty program lies in a single contigous fault location [1], even though in reality the bug diagnoses may lie in multiple contiguous fault locations [13]. Hence, a debugger may be evaluated as effective if it locates one of such locations, instead of all possible locations. Table I highlights sample diagnosis under this assumption, as well as *without* the assumption. It shows a program with faults in multiple different locations (line 6, 8 and 10), albeit under this assumption, it is sufficient to evaluate the effectiveness of a debugger with any one of such locations (e.g., line 6). In this work, we evaluate the impact of this assumption on debuggers and developer productivity. Indeed, we show that this assumption is highly prevalent in debugging literature (*see* Section III), but developers find it to be unsound and severe in practice (*see* Section VII).

B. Research Questions

- **RQ1 Prevalence of Debugging Assumptions:** How *prevalent* are the three debugging assumptions we study in automated debugging literature and bug datasets?
- **RQ2 Measured AFL Effectiveness:** What is the effect of each of the studied debugging assumptions on the measured effectiveness of AFL techniques?
- **RQ3 Proxy Effect on APR techniques:** Do these assumptions impact the measured effectiveness of APR tools?
- **RQ4 Impact on Professional Developers:** What is the effect of the studied assumptions on the productivity of professional developers in debugging practice? Specifically, we investigate the following three sub-questions:
  1) **RQ4(a) Usefulness & Closeness:** How *useful* are debugging diagnoses *with* vs. *without* each assumption? How close is the developer's diagnosis to the diagnosis provided *with* vs. *without* an assumption?
  2) **RQ4(b) Preference:** Given two diagnoses, one *with* the debugging assumption and the other *without* the assumption, which diagnosis do developers prefer?
  3) **RQ4(c) Soundness & Severity:** What debugging assumptions are the most or least important (practical) and severe (impact developer's productivity)?

Table III
PREVALENCE OF DEBUGGING ASSUMPTIONS IN THE LITERATURE, SIGNIFICANTLY HIGHER PREVALENCE (≥10%) ARE IN BOLD ("SETT." = DEBUGGING EVALUATION SETTINGS, "WITH" = WITH ASSUMPTION, "W/O" = WITHOUT ASSUMPTION)

	# (% of) Experiments involving Debug. Assumption			
Sett.	**All**	**PBU**	**Fix Location**	**Single Fault**
With	**113 (55.4%)**	41 (47.67%)	**44 (75.86%)**	28 (46.67%)
W/o	91 (44.6%)	45 (52.33%)	14 (24.14%)	32 (53.33%)

III. PREVALENCE OF ASSUMPTIONS

A. Evaluation Setup

We describe our experimental setup for evaluating the *prevalence* of the three debugging assumptions.

Experimental Approach: To determine the prevalence of each assumption (**RQ1**), we perform *data analysis* and *manual in-depth study* of the literature and bug datasets (*see* Figure 1).

Publication Collection: In this study, we collected 212 publications on automated debugging published in the last five and a half years (from 2017 till mid (June) 2022). These papers were gathered from seven (7) top-tier (core A/A*) software engineering venues, including four (4) main conferences (namely, ICSE, FSE, ASE and MSR) and three (3) journals (namely, TSE, EMSE and TOSEM). To collect these papers, we employed keywords that are relevant to automated debugging research based on the IEEE Standard Glossary of Software Engineering Terminology. We searched for 13 keywords in the online repositories and proceedings of each venue, namely "debug", "fix", "repair", "fault", "localize", "spectrum", "statistical", "bug", "locate", "localization", "fault localization", "automated debugging", "program repair".

Literature Validation and Analysis: To validate that the collected papers involve experimental evaluation of AFL/APR methods, we proof-read each paper. After validation, we filtered out 158 papers that are unrelated to automated debugging of software systems, e.g., because they are focused on fault detection or defect classification or hardware systems. After filtering and validation, we had a corpus of 63 research papers (from 2017 to mid (June) 2022) on automated debugging for our analysis. Table II provides details of the resulting papers. Most (91% of) publications are AFL research/technical papers. For each paper, we analyzed the methodology and experimental dataset to determine if and how they use each of the three aforementioned debugging assumptions. Table III shows the distribution of debugging assumptions in the literature.

Bug Datasets: Table IV provides details of the bugs and programs employed in our experiments. We employ four bug datasets to examine the prevalence of each debugging assumptions in the bug datasets, namely CoreBench [46], Siemens SIR [47], IntroClass [48] and Codeflaws [49] benchmarks. We used these datasets because of the variance of the type of bugs they contain (real, seeded and mutated bugs) and the varying complexity and maturity of their programs. These datasets are popularly used for debugging evaluations [1, 50]

Table IV
DETAILS OF SUBJECT PROGRAMS USED IN OUR EXPERIMENTS

Benchmark	#Tools	Avg. LoC	#Bugs	#Fail. Tests	#Pass. Tests	Prog. Lang.
SIR	6	148.0	74	5876	139306	C
IntroClass	6	16.5	17	129	291	C
Codeflaws	20	17.6	231	2854	8058	C
CoREBench	14	664.5	30	30	1338	C
Total	46	212	352	8889	148993	-

Analysis of Bug Datasets: For each bug dataset, we count the number of bugs that are influenced by each debugging assumption. We report the number of bugs that influence a debugging evaluation because of an assumption as "*with assumption*" and the number that may not influence an evaluation as "*without assumption*". Table V reports the results of the prevalence of each assumption in our sampled bug datasets.

Specifically, for the PBU assumption we inspect the number of bugs that have single faulty statement (as "*without assumption*"), and the number of bugs with multiple faulty statements (as "*with assumption*"). This is because the results for a debugging evaluation remains the same with or without the PBU assumption if there is only one single faulty program statement. Hence, we will count such a bug as one that does not influence the assumption (i.e., *without assumption*). Meanwhile, a debugging evaluation is impacted if there are multiple faulty statements and PBU is assumed, such bugs are counted to influence the evaluation (i.e., *with assumption*). Similarly, a debugging evaluation will not be influenced by the "fix location" assumption if the fix location of the bug is the same as the root cause location. Thus, we count such as not influencing the assumption (i.e., *without assumption*). Otherwise if the fix location and root cause do not intersect, then the assumption may influence an evaluation, hence it is counted as (i.e., *with assumption*). Finally, for the "single fault location" assumption, we check for the number of bugs that have multiple contiguous fault locations which influence the debugging evaluation for this assumption and report them as *with assumption*. Then, since bugs with a single contiguous fault location do not impact a debugging evaluation for this assumption, they are counted as *without assumption*.

Prevalence Analysis Data: Our artifact [51] ("Prevalence_Analysis") provides the data for our prevalence analysis.

B. Results: ***RQ1 Prevalence of Debugging Assumptions***

Debugging Literature: Results show that *the studied assumptions are highly prevalent in debugging literature*. Table III shows that majority (55%) of the experiments reported in the surveyed publications make at least one of the three (3) assumptions. Among the reported debugging experiments found in the surveyed papers, we observed that *the most prevalent (76%) assumption is the use of "fix location as substitute root cause diagnosis" (aka "Fix Location")*. Similarly, "PBU" and "Single Fault Location" assumptions were present in almost half (47 to 48%) of all reported experiments in the literature. These results show the high prevalence and relevance of these debugging assumptions in the research community. This finding informs the need to study their impact on the measured effectiveness of debuggers and developer's productivity.

Table V
PREVALENCE OF DEBUGGING ASSUMPTIONS IN SAMPLED BUG DATASETS, SIGNIFICANTLY HIGHER PREVALENCE (≥10%) ARE IN BOLD ("SETT." = DEBUGGING EVALUATION SETTINGS, "WITH" = WITH ASSUMPTION, "W/O" = WITHOUT ASSUMPTION)

	# (% of) Bugs fulfilling Debug. Assumption			
Sett.	**All**	**PBU**	**Fix Location**	**Single Fault**
With	471 (49%)	**284 (81%)**	136 (51%)	51 (14%)
W/o	499 (51%)	68 (19%)	130 (49%)	**301 (86%)**

The studied assumptions are highly prevalent in the literature: 55% of experiments in the surveyed publications make at least one of the three (3) assumptions.

Bug Datasets: This experiment demonstrates that *almost half (49%) of the bugs in the sampled datasets are influenced by at least one of the three studied debugging assumptions*. Indeed, these debugging assumptions are prevalent in the bug datasets – about half of the bugs in our dataset are impacted by at least one of the assumptions. Inspecting the reported debugging experiments found in the studied papers, we observed that *the PBU assumption is the most prevalent (81%), while the "single fault location" assumption is the least prevalent*. These results demonstrate the relevance of these assumptions for debugging evaluations. This shows how much these assumptions impact experiments conducted using typical bug datasets and informs the need to study their effect on debuggers and developers in practice.

About half (49% of) the bugs in the bug datasets are impacted by at least one of the debugging assumptions.

IV. EXPERIMENTAL APPROACH

Controlled Experimentation: The main research method employed in this work is *controlled experimentation* (aka split or A/B testing) [21, 20]. All research questions (except **RQ1**) involved controlled experiments with debuggers and developers. Controlled experimentation is widely used to test new features in software companies (e.g., Netflix and Google) to guide product development and data-driven decisions [52].

In this work, we employ controlled experimentation to investigate if a specific debugging setting improves debugging productivity when provided to developers, or *not* (Section VII). The *key insight* is to perform controlled experimentation with debuggers and developers under different debugging settings involving each assumption. For example, we employ this method to assess the performance of debuggers under the different debugging assumptions, in their *presence versus* their *absence*. The intuition is to evaluate the practical utility of debuggers by assessing the measured effectiveness of AFL techniques and impact on developers when the debugging assumption is present (*with*) or absent (*without)*. This allows to understand the impact of these three assumptions in practice.

Experimental Workflow Figure 1 illustrates our experimental workflow. It shows how we employ *controlled experimentation*

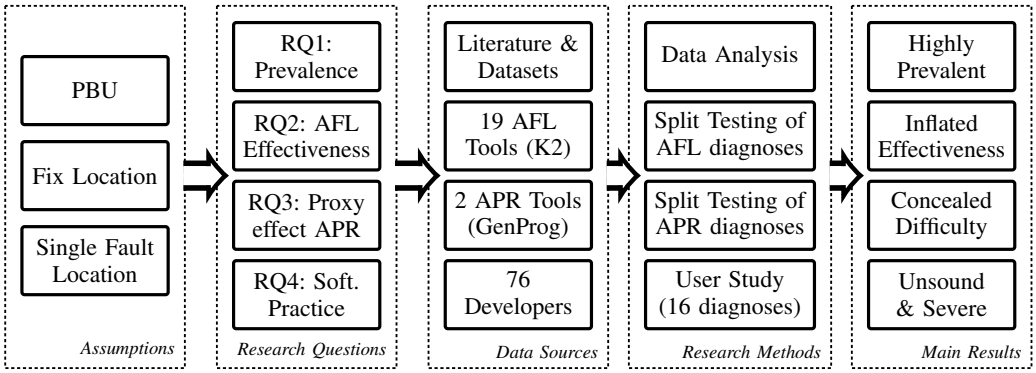

Figure 1. Experimental Workflow showing how we assess the impact of the studied debugging assumptions ("Soft." means "Software")

to study the three assumptions for all research questions (except **RQ1**). In **RQ2**, we conduct controlled experiments *with* versus *without* each assumption (using 19 AFL techniques) to determine the effect of each assumption on the measured effectiveness of AFL techniques. Results show that, indeed, these assumptions inflate the measured effectiveness of debuggers (Section V). **RQ3** studies the proxy effect of the assumptions on APR techniques using controlled experimentation. It investigates the performance of two state-of-the-art APR techniques *with* versus *without* each assumption. Section VI shows that these assumptions conceal the difficulty of both bug diagnosis and bug fixing. Finally, we conducted controlled experimentation in **RQ4** by providing developers with several debugging diagnoses *with* versus *without* the assumptions to determine the *soundness*, *severity* and *usefulness* of these assumptions in practice. We found that these assumptions are *unsound* for debuggers in practice and have a *severe effect* on developer productivity (Section VII).

V. Effect on AFL Techniques

A. AFL Evaluation Setup

Let us describe our experimental setup for this experiment. **AFL Techniques** We have chosen the two most popular AFL techniques, namely *program slicing* and *statistical debugging*. In the following, we describe each technique.

Dynamic Slicing: A *dynamic slice* [53, 54] captures all statements that are *definitely* involved in computing the values that are observed at the location where a failure is observed for a failing input. In this work, the dynamic slice consists of all statements that are reachable from a slicing criterion in the Dynamic Dependence Graph (DDG) for the failing input.

Statistical Debugging: *Statistical debugging* associates the execution of a particular program element with the occurrence of failure using *suspiciousness measures* [55, 56, 57]. Program elements (*e.g.* statements) that are observed more often in failed executions than in correct executions are deemed as more suspicious. The intuition is that a program element with a high suspiciousness score is more likely to be related to the root cause of the failure. In this paper, we employ four sets of measures consisting of 18 statistical fault localization formulas namely *human-generated* optimal measures (two (2) DStar (D^*) formulas, Wong1, Russel_Rao, Binary, Naish1 and Naish2 [7, 6, 10, 58, 59]), *popular* measures (Tarantula, Ochiai, and Jaccard [60, 61, 62, 63]), *genetic programming* (GP) evolved measures (GP02, GP03, GP13 and GP19 [64]) and *single bug optimality* measures (M9185, Kulczynski2 LexOchiai and Pattern-Similarity). These measures have been empirically evaluated to be the best performing statistical fault localization formulas [65, 66, 67, 68, 69]. They have also been shown to improve the effectiveness of program repair [3, 70].

Bugs and Programs: Experiments were conducted with a large variety of bugs and programs from the collection previously described in Section III and Table IV. These bugs include real and synthetic bugs introduced by developers, students, fault seeding and coding competitors.

Fault Locations: In our evaluation, we use two types of fault locations, the *developer-provided ground truth* fault location and the *bug fix* which are commonly used in debugging evaluations [1] (Section III). For the ground truth, we use the data provided in DBGBENCH [13], which contains fault locations provided by actual developers while debugging.

Metrics and Measures used in our evalutation include:

Suspiciousness Ranking: All fault localization techniques presented in this paper produce a *ranking*. To rank statements for *statistical debugging*, statements are listed in the order of their suspiciousness as computed by the statistical debugging formula. Meanwhile, to rank statements for *dynamic slicing*, we rank first those statements in the slice that can be reached from the slicing criterion along one backward dependency edge, then along two backward dependency edges, and so on.

Effectiveness Measures: We report effectiveness measures using the popular wasted effort metrics used in debugging evaluation [1, 71]. For instance, using the PBU assumption, the fault localization effectiveness of our AFL techniques was measured as the proportion of statements that do *not* need to be examined until finding the first fault. Assuming S_{first} is the proportion of statements that needs to be examined before the *first* faulty statement is found, under PBU, effectiveness *score* = $1 - S_{first}$. However, without PBU, we assume S_{all} is the proportion of statements that needs to be examined before *all* faulty statements are found, effectiveness *score* = $1 - S_{all}$.

Effectiveness Score Aggregation: We evaluate the overall performance of all AFL techniques for each experimental factor to show a general effect of the factor without bias

Table VI
DETAILS OF THE IMPACT OF DEBUGGING ASSUMPTIONS ON THE MEASURED EFFECTIVENESS OF AFL TECHNIQUES, SIGNIFICANT DECREASE (≥10%) IN THE MEASURED EFFECTIVENESS OF AFL TECHNIQUE(S) ARE IN **BOLD** TEXT (K2 = KULCZYNSKI2)

Debugging Assumption	**Measured AFL Effectiveness**								**Percentage Reduction in Measured AFL Effectiveness**			
	***with* Assumption**				***without* Assumption**							
(#bugs)	**ALL (19)**	**Slicing**	**K2**	**NAISH1**	**ALL (19)**	**Slicing**	**K2**	**NAISH1**	**ALL (19)**	**Slicing**	**K2**	**NAISH1**
PBU (68)	0.814	0.814	0.832	0.823	0.726	0.769	0.727	0.726	**12.2%**	5.8%	**14.5%**	**13.3%**
Fix Loc. (143)	0.706	0.746	0.686	0.686	0.646	0.724	0.620	0.619	9.2%	0.5%	5.8%	5.7%
Single Fault (51)	0.853	0.838	0.876	0.863	0.736	0.782	0.737	0.736	**15.9%**	7.2%	**19.0%**	**17.4%**
All	0.808	0.807	0.823	0.814	0.714	0.635	0.599	0.598	**13.2%**	**27.0%**	**37.5%**	**36.1%**

towards a particular AFL technique. We aggregated scores by taking the mean (average) of all scores obtained by all techniques. This dampens the effect of highly effective or ineffective AFL techniques on our results and conclusions.

Implementation Details & Platform: Both dynamic slicing and statistical debugging are implemented in 7.9 KLoC of python code. Dynamic slices are computed using FRAMA-C[72], GCOV, GIT-DIFF, GDB, and several Python libraries including PYGRAPHVIZ[73], NETWORKX[74], and MATPLOTLIB[75]. Statistical debugging was implemented using several standard command line tools, this includes GCOV[76], GIT-DIFF[77] and GDB[78]. All experiments were conducted in a Docker container running Debian GNU/Linux. The container was running on a Dell Precision 5570 with a 20-core 4.6GHz Intel Core i7-12700h CPU and 16GB of main memory.

AFL Experimental Data are provided in a Docker image in our artifact [51] ("AFL_APR_Experiments").

B. Results: *RQ2 Measured AFL Effectiveness*

RQ2 Effect on AFL techniques: We investigate the effect of the three assumptions on the measured effectiveness of AFL techniques using split testing (i.e., controlled experimentation) as described in Section IV. We report the mean effectiveness of all AFL techniques and the three best-performing AFL techniques in our experiments, i.e., Naish2, Kulczynski2 and dynamic slicing. Table VI illustrates the results of this experiment for all three debugging assumptions.

Our evaluation results show that each of *the three debugging assumptions inflate the measured effectiveness of AFL techniques by up to 38%*. We observed that there is a decrease in the performance of all 19 evaluated AFL techniques when comparing the debugging settings *with* debugging assumption versus *without* assumptions. On average, *the three studied debugging assumptions inflate the measured effectiveness of AFL techniques by 13%*. For instance, Table VI demonstrates that the measured effectiveness of the best-performing statistical debugging approach in our evaluation (i.e., Kulczynski2 formula (K2)) reduced by 38% in the presence of the assumptions (*with* assumption) when compared to the absence of the assumption (*without* assumption). This implies that a developer would need to inspect about two-fifth (38%) more program statements to locate the fault without the assumption versus with the assumption. Inspecting the effect on specific assumption, we observed that the difference in the debuggign evaluation setting as the most efffect on the PBU and Single fault location assumption with about 12% and 16% reduction in measured effectiveness between both settings. Meanwhile, the use of fix location as substitute root cause as the least effect on the performance of all approaches, yet with about nine percent inflation in measured effectiveness. Overall, these results show that the measured effectiveness of AFL techniques is strongly impacted by these assumptions. These results suggest that an AFL technique evaluated using any of these assumptions presents an inflated effectiveness measure. Hence, it is important to measure the effectiveness of AFL techniques without these assumptions to obtain more realistic effectiveness measures.

The studied debugging assumptions inflate the measured effectiveness of AFL techniques by up to 38%, on average.

VI. PROXY EFFECT ON APR TECHNIQUES

A. Evaluation Setup

APR Tools & Bug Dataset: In this experiment, we employ two APR tools, namely GENPROG [2] and ANGELIX [79]. These are two state-of-the-art, matured APR tools that have been applied in several APR studies [80, 81, 82]. We also employ two main bug datasets, namely INTROCLASS [48] and CODEFLAWS [49], we execute GENPROG on both datasets, and we execute ANGELIX only on the CODEFLAWS dataset. To avoid biasing our experiments, we do not run ANGELIX on the INTROCLASS dataset. This is because it requires out-of-the-box manual modifications of the ANGELIX tool-chain to address the compilation and test oracle requirements of INTROCLASS. Firstly, one will need to transform each INTROCLASS program or update ANGELIX's macro conversion for compilation.[1] Secondly, one will need to manually generate an assertion file for each tool in the INTROCLASS benchmark, since correctly determining if a test has passed/failed requires comparing the program output to a manually defined expected string output.[2] We avoid conducting such out-of-the-box manual modifications of ANGELIX to mitigate experimental bias.

Effectiveness Measures: We employ *four* (4) well-known effectivness metrics to determine the measured effectiveness of APR tools, namely:

[1] ANGELIX requires converting "`printf`" statements to macros, including formatted versions of "`printf`", e.g., "`sprint`", "`fprintf`", etc. This also requires (automatic) derivation of the variable type in such statements.

[2] ANGELIX does not support a test oracle that compares a program's behavior versus a reference implementation, which is necessary for INTROCLASS.

Table VII
DETAILS OF THE PROXY EFFECT OF DEBUGGING ASSUMPTIONS ON THE MEASURED EFFECTIVENESS OF APR TECHNIQUES ((R) = NUMBER OF REPAIRS)

Tool	Debugging Assumption	Measured APR Effectiveness *with* Assumption				Measured APR Effectiveness *without* Assumption				Percentage Reduction in Average Measured APR Effectiveness			
		#gens	#Runs	Time (s)	Ⓡ	#gens	#Runs	Time (s)	Ⓡ	#gens	#Runs	Time (s)	Ⓡ
GenProg	PBU	1.25	862.37	36.24	8	1.90	1392.20	147.22	10	52.0%	61.4%	306.2%	25.0%
	Fix Location	0.00	18.00	0.33	2	3.18	1433.54	129.19	11	0%	7864.1%	39286.9%	450%
	Single fault	1.17	1027.83	46.75	6	1.71	1746.57	207.72	7	46.9%	69.9%	344.3%	16.7%
Angelix	PBU	7.33	19.00	19.10	3	8.00	21.67	22.26	3	9.1%	14.0%	16.5%	0%
	Fix Location	8.50	23.50	26.52	2	4.81	46.52	132.54	21	-43.4%	98.0%	399.8%	950%
	Single fault	7.33	19.00	19.10	3	10.00	23.00	26.96	2	36.4%	21.0%	41.1%	-33.3%
Total	All	25.58	1969.70	148.04	24	29.60	4663.50	655.89	54	15.7%	136.8%	343.0%	125%

- **Number of Repair Generations (called "#gen"):** This is the number of generations ("evolution" rounds) made by the APR tool before the correct patch was generated (i.e., a patch passing all tests in the test suite). Specifically, for GenProg, this refers to the number of generations of the genetic algorithmic search when searching for a repair [2]. For Angelix, "#gen" refers to the number of path counts called "angelic paths" [79]. For instance, Table VII (row 1) shows that, with the "PBU" assumption, GenProg generates eight (8) program repairs (#repairs Ⓡ) after about 1.25 evolutions/generations (#gens=1.25), on average. Likewise, Table VII (row 2) shows that, with the "fix location" assumption, GenProg generates two (2) repairs (#repairs Ⓡ) in the initial generation, without any genetic evolution/generation (#gens=0), on average.
- **Number of Program executions (called "#Runs"):** This refers to the number of times the program is executed using the entire test suite to validate and determine the correctness of the APR generated patches.
- **Execution time (called "Time"):** This is the time taken before a correct repair was generated by the APR tool for the bug.
- **Number of Correct Repairs (called "#repairs"):** This refers to the number of buggy programs that were correctly repaired by the APR tool.

Debugging Diagnosis Settings: In this experiment, we examined the measured effectiveness of APR tools using three different debugging settings. We used the perfect debugging diagnosis (i.e., set of candidate fault locations for repair) from the bug dataset for each setting. This is the best candidate location to fix the bug, e.g., the exact fix location based on the ground truth from the bug dataset.

Research Protocol: For each assumption, each buggy program and each type of diagnosis in our dataset, we evaluate the effectiveness of the APR tool *with* and *without* the debugging assumption at hand.Then, we collect the measured effectiveness metrics aforementioned and compute the *difference* in the measured effectiveness in both settings. For a balanced evaluation, all experiments for GenProg and Angelix were conducted with a time budget of 30 minutes and a limit of 15 repair generations, on the same host system.

Debugging Assumptions Settings: For the PBU assumption, we use faults with multiple faulty statements in one contiguous location and execute each APR tool (e.g., ANGELIX) with a "single faulty statement" versus "all faulty statements". To evaluate the "fix location" assumption, we employed the "actual fix location" (in the benchmark) versus the "root cause location" as the patch location provided to each APR tool, using faults with no intersection between root cause and fix locations. All root causes were manually determined by developers involved in previous studies (DBGBENCH [13]/COREBENCH [46]) and the rest (e.g., INTROCLASS, SIR, CODEFLAWS) were manually determined by two of the authors/researchers of this study. Meanwhile, for the single fault assumption, we have employed a set of faults with multiple contiguous fault locations and provided "a single contiguous fault location" versus "all contiguous fault locations" as the patch location. For this assumption, we use the fault locations provided by the benchmark.

APR Experimental Data are provided in a Docker image in the paper's artifact [51] ("AFL_APR_Experiments").

*B. Results: **RQ3 Proxy Effect on APR Effectiveness***

RQ3 Proxy Effect on APR techniques: This experiment examines if the studied assumptions have a "proxy" effect on the measured effectiveness of APR tools. This is particularly important since AFL is often the first step of repair, and many APR tools use AFL tools to determine the fault locations for fixing bugs. Table VII shows our results for this experiment.

Our evaluation results show that *these assumptions reduce the measured effectiveness of APR tools by 2X, and conceal the difficulty of fixing programs by up to 3X.* On one hand, Table VII shows that the number of completed program repairs (Ⓡ) is (125%) larger *without* the assumptions, than *with* the assumptions. This implies that the use of these assumptions impact the measured effectiveness of APR tools. Evaluating with these assumptions may make an APR tool seem less useful than it truly is in practical settings, especially in terms of the number of the resulting repairs. On the other hand, we also observed that under these assumptions, the true cost of fixing the bugs in terms of the number of generations (#gen), the number of program runs (#Runs) and the time taken is masked by these assumptions (*see* Table VII). We observed that both repair effectiveness and the cost of repair are higher *without* these assumption than *with* these assumptions. APR

tools are indeed more effective in producing repairs than the assumptions make them appear, albeit they are also more expensive without these assumptions. These findings inform the need to employ AFL appropriately for APR evaluations.

Debugging assumptions reduce the measured effectiveness of APR tools: 125% more repairs are generated without these assumptions with repair time being up to 3X higher.

VII. Impact on Professional Developers

A. User Study Design

We conducted a user study to evaluate the *impact of these debugging assumptions on developers and their productivity*. The goal is to examine if debugging diagnosis provided *with* and *without* each of the three main assumptions influences the productivity of developers. In particular, we ask developers about the *usefulness*, *preference*, *soundness* and *severity* of these assumption in debugging practice.

Bug Datasets: Our user study involved 76 developers debugging eight (8) buggy programs using 16 debugging diagnoses.

Debugging Assumptions Settings: The user study was conducted using the debugging settings described in Section VI ("Debugging Assumptions Settings"). To evaluate the PBU assumption, we compare participants' responses to a diagnosis consisting of a "single faulty statement" versus "multiple faulty statements". We evaluate the "fix location assumption" by comparing participants' responses for the diagnoses with an actual "fix location" versus a "root cause location". For the "single fault" assumption, we compare participants' responses for "a single fault location" versus "all fault locations".

Study Questionnaire This user study is organized into three main parts corresponding to the three main aforementioned research questions. Specifically, the questionnaire for this user study was divided into the following parts: (a) *Usefulness & Closeness* which examines how developers perceive the usefulness of these three debugging assumptions, and rate the closeness of the provided diagnosis to the developer's own (perfect) diagnosis. (b) *Preference* where we ask developers their preference for each debugging diagnosis with or without the assumption. (c) *Soundness & Severity* evaluates the soundness of these assumption for debuggers and the severity of these assumptions on developer's productivity in practice.

The first and second part of the study were conducted without identifying or describing the diagnosis or bug associated with a debugging assumption. Only the third part involved the description of the assumptions. In summary, the entire study questionnaire contains about 100 questions.

Pilot Study: First, we conducted a pilot user study involving seven (7) software engineering professionals/researchers. The goal of the study is to obtain early feedback on the study, i.e., find errors in the questions or study design and spot unclear/(mis)leading questions. In addition, we wanted to determine an estimated duration and compensation for the study. Participants of the pilot study were all researchers, PhD students and postdoctoral researchers. We note that participants of the pilot study were not compensated. The Pilot study was conducted in June, 2022 and lasted for about a month. For each participant in the pilot study, we conducted an interview (15 to 30 minutes) to obtain feedback on the study design, unclear questions, and their understanding of the questions.

After the pilot study, we modified the study design and clarified some questions. For instance, we added the description of certain terms, e.g., soundness and severity for clarity. Based on the feedback from the pilot study, we also added a progress tab as well as transition sections describing each part of the study before participants begin answering questions. We also included new questions based on the pilot feedback, e.g., questions about other debugging assumptions or concerns of developers about assumptions. Using the response time of participants, we determined that the study takes about 40 to 90 minutes depending on C expertise and familiarity.

Recruitment and Compensation: This study was advertised on three major online platforms, namely Amazon Mturk[83], LinkedIn [84] and Prolific [85]. These are popular online platforms for recruiting professional software developers for research studies. We also advertised the study on certain professional C developers mailing lists and communities. Participants were compensated with about $20 for completing the study within three hours. The main study took six weeks starting in in July 2022. We collected 160 unique responses. After response validation, we filtered out 84 responses for failing most of the validation questions (e.g., cause/fix of the bugs). Overall, we had 76 valid responses from the study.

Demography: In this study, we had a total of 76 unique participants with valid responses. Most (58%) of the participants are professional software developers or testers, about 30% are students and 12% are researchers. About half (49%) of the participants have three or more years of software development experience, and about a third (34%) have more than one year experience in software development. Three out of four of the respondents have intermediate to advance level of skill in C programming and only one in four are novices. Almost half of our participant indicate that they most often program in C.

Response Data Validation: Our study contains several questions that allow to validate the correctness of participant's responses and the proficiency of participant to take the study and be compensated for it. For instance, we ask developers to provide the cause of the bug and describe how they would fix the bug. We also repeat certain questions, albeit re-phrased differently or at different sections of the study (e.g., about the root cause of the bug and how to fix the bug). There were about 10 validation questions designed to ensure collected responses are indeed valid responses from professional software developers. For each submitted response, we had checked the correctness of the validation questions and only accepted responses with at least 75% correctness. These questions allowed to exempt (84) invalid responses (e.g., from people with low C or software development experience).

Response Data Analysis: For quantitative questions (e.g.,

preferred diagnosis or level of soundness), we analyze all of the responses in terms of the frequency of responses for all possible options. Meanwhile, we employed a coding protocol [86] to analyze for qualitative analysis of free-text questions (e.g., why an assumption/diagnosis is realistic). Our protocol involved two researchers who independently coded and categorized all responses into specific classes. Then, both researchers meet to resolve conflicts in the coding and agree on the categorization of the responses.

User Study Data: Our artifact [51] ("User_Study") contains the user study questionnaire, responses and analysis.

B. User Study Results

RQ4(a) Usefulness & Closeness: This experiment evaluates how developers perceive the usefulness of debugging diagnoses produced under two different evaluation settings, namely in the absence (aka *without*) versus presence (aka *with*) of each of the three studied debugging assumptions. We also evaluate how developers rate the closeness of the debugging diagnoses produced for each setting to developers' own perfect diagnosis for the bug. These experiment involved a total of six debugging settings, i.e., two bug diagnoses representing with and without an assumption for three real-world buggy programs that characterize each assumption. All diagnoses and bugs were provided without reference to the debugging assumption in the setting to avoid biasing the responses. We do not identify or describe the diagnosis or the bug associated with the debugging assumption for each diagnosis or bug. Table VIII highlights the results of this experiment.

Usefulness: Our user study results show that for all three assumptions, *developers find the debugging diagnoses provided without an assumption to be more useful than diagnoses with the assumption*. Table VIII highlights that most developers (47%) find the debugging diagnoses without the debugging assumptions to be *most useful*, and majority (80%) of developers find this setting to be *mid to most useful*. In contrast, most developers (44%) perceive the debugging setting with the debugging assumptions to be *least useful*, and the majority (63%) of developers find the diagnoses in this debugging setting to be *mid to least useful*. Developers rated the diagnoses *with* each debugging assumption lower than *without* the assumption for all three assumptions (*see* Table VIII). For instance, almost half (47%) of the participants rated the diagnoses provided *without the single fault assumption* to be most useful. Meanwhile, the diagnoses *with the single fault assumption* were rated mid-level or least useful by most (about 72% of) developers. These results emphasize the negative impact of the assumptions on the usefulness of debugging diagnoses. It shows that each of these debugging assumptions reduced the usefulness of debugging diagnoses for developers.

Diagnoses provided without the assumptions are considered mid to most useful by majority of developers (80%) while, most (63% of) developers find the diagnoses provided with the assumptions to be mid to least useful.

Closeness: Developers *rated the diagnoses provided without each of the three debugging assumptions to be the closest to their own perfect diagnoses*. These diagnoses (without the assumptions) are the most similar to the root cause diagnoses of developers. Most developers (55%) consider the diagnoses under any of the three assumptions to be *mid to least closest* to their diagnosis. Meanwhile, majority of participants (81%) consider the diagnoses *without* the assumptions to be *mid to most closest* to their diagnosis. This is particularly prominent for the *single fault location assumption* where majority (83%) of developers find the diagnoses *without the assumption (i.e., providing multiple fault location diagnosis)* to be *mid to most closest* to their own perfect diagnoses. In contrast, about one-third of developers (32%) think the diagnoses *with* the single fault assumption is *least closest* to their own diagnoses. Most developers (67%) consider the diagnoses provided under the *single fault assumption* to be *mid to least closest* to their own perfect diagnoses. Overall, the diagnoses provided under these assumptions are different from developers' desired diagnoses.

Diagnoses provided without the assumptions are the closest to developer's own perfect diagnosis. Most developers (83%) consider the diagnosis with the single fault assumption to be mid to least close to their own.

RQ4(b) Preference: We evaluate the debugging diagnosis developers will prefer for each assumption. Without identifying or explaining the assumptions, we provide buggy programs with two different diagnoses representing the presence and absence of an assumption. Then, we ask developers which diagnoses they prefer for each bug. Table IX provides details of developers' preference for each debugging assumption.

We found that *developers prefer debugging diagnoses provided without the debugging assumptions than with the assumptions*. Table IX shows that the debugging diagnoses provided without these assumptions are almost twice as desirable (or 96% more desirable) for developers than diagnosis under the assumptions, on average. For instance, developers prefer the debugging diagnoses for multiple fault locations 5X as much as with single fault location assumption. These results show that the debugging diagnoses provided under current evaluation settings (i.e., "*with* assumptions") are least desired by developers. In fact, developers strongly prefer the alternative debugging settings without these assumptions.

Four times more developers preferred debugging diagnoses without the assumption than the diagnosis provided with the assumption.

RQ4(c) Soundness and Severity: This experiment examines the level of soundness and severity of the studied debugging assumptions. We asked developers to rate the level of soundness and severity of each assumption, from the most to least sound/severe. "Soundness" refers to the degree to which the assumption holds for a debugger (i.e., realism) in practice. Meanwhile, "Severity" refers to the impact of the assumption

Table VIII
USEFULNESS OF DEBUGGING DIAGNOSIS AND CLOSENESS OF THE DEBUGGING DIAGNOSIS TO THE DEVELOPER'S DIAGNOSIS, HIGHEST SCORES ARE IN **BOLD** TEXT ("'WITH" AND 'W/O" MEANS THE DEBUGGING SETTINGS "WITH" AND "WITHOUT" AN ASSUMPTION)

	Level	Usefulness				Closeness			
	(Likert)	PBU	Fix Location	Single Fault	All	PBU	Fix Location	Single Fault	All
With	**Least** (0-1)	11 (14.47%)	5 (6.58%)	27 (35.53%)	43 (18.86%)	11 (14.47%)	8 (10.53%)	24 (31.58%)	43 (18.86%)
	Mid (2-3)	**35 (46.05%)**	**37 (48.68%)**	**28 (36.84%)**	**100 (43.86%)**	21 (27.63%)	33 (43.42%)	**28 (36.84%)**	82 (35.96%)
	Most (4-5)	30 (39.47%)	34 (44.74%)	21 (27.63%)	85 (37.28%)	**44 (57.89%)**	**35 (46.05%)**	24 (31.58%)	**103 (45.18%)**
W/o	**Least** (0-1)	15 (19.74%)	14 (18.42%)	17 (22.37%)	46 (20.18%)	17 (22.37%)	14 (18.42%)	13 (17.11%)	44 (19.30%)
	Mid (2-3)	23 (30.26%)	28 (36.84%)	23 (30.26%)	74 (32.46%)	17 (22.37%)	26 (34.21%)	30 (39.47%)	73 (32.02%)
	Most (4-5)	**38 (50.00%)**	**34 (44.74%)**	**36 (47.37%)**	**108 (47.37%)**	**42 (55.26%)**	**36 (47.37%)**	**33 (43.42%)**	**111 (48.68%)**

Table IX
DEVELOPERS' PREFERENCE FOR DEBUGGING ASSUMPTIONS, HIGHER PREFERENCE AND SIGNIFICANT IMPROVEMENTS (≥25%) ARE IN **BOLD** ("ASSUM." = "ASSUMPTION", "IMPR." = "IMPROVEMENT OF 'WITHOUT' ASSUMPTION OVER 'WITH' ASSUMPTION")

Debugging Assumption	(#) % of Preferred Diagnosis		
	With Assum.	Without Assum.	Impr.
Single Fault Loc.	(13) 17.11%	**(63) 82.89%**	**385%**
PBU	(29) 38.16%	**(47) 61.84%**	**62%**
Fix Location	(35) 46.05%	**(41) 53.95%**	17%
All	(77) 33.77%	**(151) 66.23%**	**96%**

on developer productivity in practice. Table X highlights the level of soundness and severity of each assumption.

We observed that *developers found the use of "fix location as substitute fault location" (aka "fix location") assumption to be the least sound and most severe debugging assumption*. Most developers (38%) believe this assumption is the *least* realistic in practice and has the highest (negative) impact on developer productivity. This is particularly concerning since "fix location" is the most prevalent (76%) assumption in debugging experiments (*see* RQ1, Section III). Results further show that the "single fault location" assumption has the least severity and soundness concern for developers. Meanwhile developers found the PBU assumption to have intermediate severity and soundness. Overall, these results imply that the most common assumption in the debugging research community is also the most unsound and severe assumption to developers. Hence, we believe this assumption is a major threat to debugging evaluation and resolving it should be prioritized.

Two-fifth (≈38%) of developers found the most prevalent "fix location" assumption to be the least sound for debuggers and most severe on developer productivity.

Discussions and Qualitative Analysis: We analyzed the free-text responses in our user study using a coding protocol (Section VII). On one hand, results show that most (60%) developers have negative concerns about the diagnosis provided under these assumptions and their soundness in practice. Most developers mention that these assumptions result in *low precision and over-simplification of bug diagnoses*. Developers also believe that these assumptions result in diagnoses that lead to *inadequate debugging information and wrong diagnosis*. Several participants mentioned that the assumptions lead to bug diagnosis that do "*not provide enough [debugging] information*" and "*waste [developer] time*". On the other hand, we observed that fewer (35% of) developers see positive benefits of the assumptions. Some participants believe that these assumptions may ease debugging and bug understanding. Participants stated that the assumption may *still* "*reduce the [debugging] work of a software developer*" and "*locate fix spots more quickly, however it can lead to mistakes*". Overall, these results inform the need to assess debuggers *without* these assumptions to improve debugger adoption/utility in practice.

Most developers (60%) believe these assumptions are unrealistic in practice because they lead to imprecise, wrong or inadequate bug diagnoses.

VIII. LIMITATIONS AND THREATS TO VALIDITY

Internal Validity: The main threat to internal validity of this work is the correctness of our implementation and user study questionnaire. To mitigate this threat, we have tested our implementation with manual tests to ensure correctness. To ensure the correctness of our user study, we conducted a pilot study and included several validation questions.

External Validity: This is the extent to which our results generalize to other objects which are not included in this study. The main external threat to validity is that all our subject programs are open-source C programs, albeit from well-known benchmarks and projects. Indeed, the bugs found in other (commercial) projects may have different features and complexity. Thus, we do not claim that our findings generalize to other software other than the ones represented in our study.

Construct Validity: This is the degree to which our experiments measures what it claims to be measuring. Notably, our *measure of fault localization effectiveness* is the main threat to construct validity. In this work, we measure debugging effectiveness as *ranking-based relative wasted effort*, such that an AFL technique is more effective the higher it ranks faulty statement(s). This is the standard metric for evaluating the performance of AFL techniques [1, 71]. To further mitigate this threat, we have also performed experiments with perfect diagnoses as provided by humans.

Survey Bias: Our prevalence analysis (RQ1) may be biased by the choice of period, SE venues and bug datasets. Thus it may not generalize to other periods (e.g., before 2017), venues (e.g., ICST, ISSTA, etc) or benchmarks (e.g., Defects4J[87]). To mitigate this threat, we have analyzed recent papers from the

Table X
LEVEL OF SOUNDNESS AND SEVERITY OF DEBUGGING ASSUMPTIONS SHOWING THE NUMBER OF PARTICIPANTS THAT RATE AN ASSUMPTION AS THE *most*, *intermediate* AND *least* SOUND OR SEVERE.

Debugging Assumption	Level of Soundness Number of Developers (%)			Level of Severity Number of Developers (%)		
	Most Sound	**Intermediate**	**Least Sound**	**Most Severe**	**Intermediate**	**Least Severe**
Single Fault	**38 (50.00%)**	13 (17.11%)	25 (32.89%)	26 (34.21%)	25 (32.89%)	25 (32.89%)
Fix Location	16 (21.05%)	31 (40.79%)	**29 (38.16%)**	**28 (36.84%)**	24 (31.58%)	24 (31.58%)
PBU	22 (28.95%)	**32 (42.11%)**	22 (28.95%)	22 (28.95%)	**27 (35.53%)**	**27 (35.53%)**

top-tier (Core A*/A) venues, and selected benchmarks that are popularly used in the community. We have also provided our experimental data such that our survey can be easily extended to other settings (e.g., venues).

User Study Bias: Our user study with developers may suffer from wrong respsonses, cognitive bias and observer-expectancy bias especially since developer's responses are self-reported and developers may behave differently during the study. We mitigate these threats by adding validation and repetitive questions to our study to vet incorrect responses. We have also avoided typical user study pitfalls such as leading questions by conducting a pilot study to determine if any of these biases are prominent in our study design. To avoid respondent priming, we only introduce the assumptions explicitly in the third part of the study with sample code, bugs and diagnoses, since developers must understand the assumption to determine its soundness and severity in debugging practice.

IX. RELATED WORK

Effectiveness of Automated Fault Localization: Several researchers have evaluated the effectiveness of AFL techniques [50, 1]. The most closely related works to this paper have focused on the *empirical evaluation of AFL techniques* with real bugs or developers. Parnin and Orso evaluated whether AFL techniques help developers debug faster [11]. The authors found that it is important to highlight multiple suspicious statements for the developer, in practice. Pearson et al. also evaluated the performance of different statistical debugging formulas on artificial and real bugs. The authors found that artificial faults are not useful for predicting which fault localization techniques perform best on real faults [9]. DBGBENCH provides a benchmark containing actual root causes of several errors from the CoREBench benchmark as determined by actual programmers [46, 13]. This benchmark allows for automatic assessment of debugging techniques on real-world errors, actual developer identified root causes and patches. Unlike our work, none of these papers evaluated the impact of the three studied assumptions on the debugging performance of AFL tools and the productivity of developers.

Experimental Factors and Debugging Effectiveness: Automated Fault Localization Techniques are often evaluated using bugs and programs collected from different benchmarks or (open source) software projects [1]. These programs often have different features that can be manipulated to improve the effectiveness of an AFL technique. Wong and Debroy describe several techniques that manipulate certain features (e.g., test suite) can improve the debugging effectiveness of AFL techniques [50]. For instance, researchers have shown that test selection and prioritization can improve effectiveness of AFL techniques [88, 89]. Other researchers have curated bugs with different defects in order to allow for the objective evaluation of automated program repair (APR) tools [49], demonstrating that different APR methods may fix certain types of faults more effectively than others. Unlike previous works, we examine the impact of experimental factors on the effectiveness of AFL techniques and developers. Similar to our findings (RQ3), Liu et al. [90] shows that AFL (configurations) may bias the results of APR tools. Liu et al. [90] evaluated the impact of fault localization (FL) configurations (e.g., levels of detection granularity (statement vs. method level)) when comparing *only* APR tools. Meanwhile, our work investigates the impact of three frequently adopted AFL assumptions on developers' debugging productivity, and the measured effectiveness of debuggers (AFL methods and APR tools).

X. CONCLUSION

We evaluated the impact of several experimental factors on the empirical effectiveness of *automated fault localization* and Automated Program Repair techniques to understand and mitigate the threats of this assumption for sound and practical evaluation. Specifically, we focused on the *perfect bug understanding* assumption, use of bug fix as substitute of root causes and single fault location assumption. We found that these factors are prevalent in debugging literature and observed that they inflate the effectiveness of AFL and underestimate the effectiveness of APR techniques, concealing the difficulty of debugging. Developers also emphasize that these assumptions are impractical and may inhibit debugging practice. More importantly, most (66% of) developers prefer debugging diagnoses without these assumptions. Our findings motivate the need for more conservative evaluations, without making these assumptions, and to assess AFL and APR techniques in a more realistic and practical settings. We provide an artifact [51] that documents our implementation and experimental data, as well as a website to ease replication, scrutiny and reuse:

https://debugging-assumptions.github.io/

XI. ACKNOWLEDGMENT

This project was funded by the Luxembourg National Research Foundation (FNR) CORE Jr C21/IS/15845400/GTDebug. Ezekiel Soremekun and Lukas Kirschner acknowledge the financial support of FNR.

REFERENCES

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

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

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

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

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

[6] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[7] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 2012, pp. 21–30.

[8] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05, 2005, pp. 342–351.

[9] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 609–620.

[10] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 1. IEEE, 2007, pp. 449–456.

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

[12] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 572–583.

[13] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "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*, 2017, pp. 117–128.

[14] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 661–673.

[15] Y. Küçük, T. A. Henderson, and A. Podgurski, "Improving fault localization by integrating value and predicate based causal inference techniques," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 649–660.

[16] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 1, pp. 1–30, 2017.

[17] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang, "Combining spectrum-based fault localization and statistical debugging: An empirical study," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 502–514.

[18] A. Nikanjam, H. B. Braiek, M. M. Morovati, and F. Khomh, "Automatic fault detection for deep learning programs using graph transformations," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.

[19] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S.-C. Cheung, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2348–2368, 2019.

[20] R. Kohavi, A. Deng, R. Longbotham, and Y. Xu, "Seven rules of thumb for web site experimenters," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 1857–1866.

[21] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann, "Online controlled experiments at large scale," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 1168–1176.

[22] Z. Ren, H. Jiang, J. Xuan, and Z. Yang, "Automated localization for unreproducible builds," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 71–81.

[23] Z. He, Y. Chen, E. Huang, Q. Wang, Y. Pei, and H. Yuan, "A system identification based oracle for control-cps software fault localization," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 116–127.

[24] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1027–1038.

[25] A. Amar and P. C. Rigby, "Mining historical test logs to predict bugs and localize faults in the test logs," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 140–151.

[26] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.

[27] M. M. Rahman and C. K. Roy, "Improving ir-based bug localization with context-aware query reformulation," in *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 621–632.

[28] Y. Lin, J. Sun, L. Tran, G. Bai, H. Wang, and J. Dong, "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*, 2018, pp. 509–519.

[29] M. Maamar, N. Lazaar, S. Loudni, and Y. Lebbah, "Fault localization using itemset mining under constraints," *Automated Software Engineering*, vol. 24, no. 2, pp. 341–368, 2017.

[30] T. S. Zaman, X. Han, and T. Yu, "Scminer: Localizing system-level concurrency faults from large system call traces," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 515–526.

[31] Z. Ren, C. Liu, X. Xiao, H. Jiang, and T. Xie, "Root cause localization for unreproducible builds via causality analysis over system call tracing," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 527–538.

[32] M. Rath, D. Lo, and P. Mäder, "Analyzing requirements and traceability information to improve bug localization," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 442–453.

[33] S. A. Akbar and A. C. Kak, "A large-scale comparative evaluation of ir-based tools for bug localization," in *Proceedings of the 17th international conference on mining software repositories*, 2020, pp. 21–31.

[34] X. Ma, S. Wu, E. Pobee, X. Mei, H. Zhang, B. Jiang, and W.-K. Chan, "Regiontrack: a trace-based sound and complete checker to debug transactional atomicity violations and non-serializable traces," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–49, 2020.

[35] J. Troya, S. Segura, J. A. Parejo, and A. Ruiz-Cortés, "Spectrum-based fault localization in model transformations," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 3, pp. 1–50, 2018.

[36] Y. Kim, S. Mun, S. Yoo, and M. Kim, "Precise learn-to-rank fault localization using dynamic and static features of target programs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–34, 2019.

[37] L. Arcega, J. Font, Ø. Haugen, and C. Cetina, "Bug localization in model-based systems in the wild," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–32, 2021.

[38] X. Li, W. E. Wong, R. Gao, L. Hu, and S. Hosono, "Genetic algorithm-based test generation for software product line with the integration of fault localization techniques," *Empirical Software Engineering*, vol. 23, no. 1, pp. 1–51, 2018.

[39] F. Feyzi, "Cgt-fl: using cooperative game theory to effective fault localization in presence of coincidental correctness," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3873–3927, 2020.

[40] N. Bayati Chaleshtari and S. Parsa, "Smbfl: slice-based cost reduction of mutation-based fault localization," *Empirical Software Engineering*, vol. 25, no. 5, pp. 4282–4314, 2020.

[41] B. Liu, S. Nejati, L. C. Briand *et al.*, "Effective fault localization of automotive simulink models: achieving the trade-off between test oracle effort and fault localization accuracy," *Empirical Software Engineering*, vol. 24, no. 1, pp. 444–490, 2019.

[42] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[43] S. Benton, X. Li, Y. Lou, and L. Zhang, "Evaluating and improving unified debugging," *IEEE Transactions on Software Engineering*, 2021.

[44] T. T. Nguyen, K.-T. Ngo, S. Nguyen, and H. Vo, "A variability fault localization approach for software product lines," *IEEE Transactions on Software Engineering*, 2021.

[45] D. Jarman, J. Berry, R. Smith, F. Thung, and D. Lo, "Legion: Massively composing rankers for improved bug localization at adobe," *IEEE Transactions on Software Engineering*, 2021.

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

[47] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria," in *Proceedings of 16th International conference on Software engineering*. IEEE, 1994, pp. 191–200.

[48] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[49] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "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, 2017, pp. 180–182.

[50] W. E. Wong and V. Debroy, "A survey of software fault localization," *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, vol. 9, 2009.

[51] E. Soremekun, L. Kirschner, M. Böhme, and M. Papadakis, "Artifact for Evaluating the Impact of Experimental Assumptions in Automated Fault Localization," 1 2023. [Online]. Available: https://figshare.com/articles/conference_contribution/Debugging_Assumptions_Artifact/21786743

[52] H. Gui, Y. Xu, A. Bhasin, and J. Han, "Network a/b testing: From sampling to estimation," in *Proceedings of the 24th International Conference on World Wide Web*, 2015, pp. 399–409.

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

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

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

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

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

[58] P. F. Russel, T. R. Rao *et al.*, "On habitat and association of species of anopheline larvae in south-eastern madras." *Journal of the Malaria Institute of India*, vol. 3, no. 1, 1940.

[59] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, p. 11, 2011.

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

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

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

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

[64] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 244–258.

[65] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, p. 31, 2013.

[66] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localisation," in *International Symposium on Search Based Software Engineering*. Springer, 2013, pp. 224–238.

[67] D. Landsberg, "Methods and measures for statistical fault localisation," Ph.D. dissertation, University of Oxford, 2016.

[68] D. Landsberg, H. Chockler, D. Kroening, and M. Lewis, "Evaluation of measures for statistical fault localisation and an optimising scheme," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2015, pp. 115–129.

[69] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating faults with program slicing: an empirical analysis," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–45, 2021.

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

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

[72] Frama-C. (2007) Frama-c - framework for modular analysis of c programs. [Online]. Available: https://frama-c.com/

[73] P. Developers. (2014) Pygraphviz. [Online]. Available: https://pygraphviz.github.io/

[74] N. developers. (2014) Networkx - network analysis in python. [Online]. Available: https://networkx.org/

[75] T. M. development team. (2012) Matplotlib - visualization with python. [Online]. Available: https://matplotlib.org/

[76] I. Free Software Foundation. (1987) gcov - a test coverage program. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[77] T. G. community. (2005) git-diff. [Online]. Available: https://git-scm.com/docs/git-diff

[78] I. Free Software Foundation. (1986) Gdb: The gnu project debugger. [Online]. Available: https://www.sourceware.org/gdb

[79] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[80] X.-B. D. Le, D. Lo, and C. Le Goues, "Empirical study on synthesis engines for semantics-based program repair," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 423–427.

[81] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust enhancement issues in program repair," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*, 2022.

[82] M. Jiang, T. Y. Chen, Z. Q. Zhou, and Z. Ding, "Input test suites for program repair: A novel construction method based on metamorphic relations," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 285–303, 2020.

[83] A. M. T. (MTurk). (2005) Amazon mechanical turk (mturk). [Online]. Available: https://www.mturk.com/

[84] Linkedin. (2003) Linkedin. [Online]. Available: https://www.linkedin.com/

[85] Prolific. (2011) Prolific. [Online]. Available: https://www.prolific.co/

[86] K. Charmaz, *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006.

[87] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[88] P. Rao, Z. Zheng, T. Y. Chen, N. Wang, and K. Cai, "Impacts of test suite's class imbalance on spectrum-based fault localization techniques," in *2013 13th International Conference on Quality Software*. IEEE, 2013, pp. 260–267.

[89] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 82–91.

[90] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 102–113.