# Iterative User-Driven Fault Localization

Xiangyu Li[1(✉)], Marcelo d'Amorim[2], and Alessandro Orso[1]

[1] Georgia Institute of Technology, Atlanta, USA
{xiangyu.li,orso}@cc.gatech.edu
[2] Federal University of Pernambuco, Recife, Brazil
damorim@cin.ufpe.br

Abstract. Because debugging is a notoriously expensive activity, numerous automated debugging techniques have been proposed in the literature. In the last ten years, statistical fault localization emerged as the most popular approach to automated debugging. One problem with statistical fault localization techniques is that they tend to make strong assumptions on how developers behave during debugging. These assumptions are often unrealistic, which considerably limits the practical applicability and effectiveness of these techniques. To mitigate this issue, we propose Swift, an iterative user-driven technique designed to support developers during debugging. Swift (1) leverages statistical fault localization to identify suspicious methods, (2) generates *high-level* queries to the developer about the correctness of specific executions of the most suspicious methods, (3) uses the feedback from the developer to improve the localization results, and (4) repeats this cycle until the fault has been localized. Our empirical evaluation of Swift, performed on 26 faults in 5 programs, produced promising results; on average, Swift required less than 10 user queries to identify the fault. Most importantly, these queries were only about input/output relationships for specific executions of the methods, which developers should be able to answer quickly and without having to look at the code. We believe that Swift is a first important step towards defining fault localization techniques that account for the presence of humans in the loop and are practically applicable.

1 Introduction

Debugging contributes greatly to software development costs [25]. It is therefore not surprising that researchers and practitioners alike invested much effort in defining techniques that can help developers in this task. Statistical fault localization (SFL) techniques, in particular, became extremely popular in recent years (*e.g.,* [4,6–8,14,15,18,22,27,28]). These techniques compute suspiciousness values for various program entities using coverage information of passing and failing test cases and use theses values to produce a ranked list of program entities in decreasing order of suspiciousness. While significant progress has been made in this field, there is evidence that (1) asking developers to examine a possibly long list of suspicious program entities in order and (2) expecting developers to recognize faulty lines by simply looking at them are both unrealistic expectations. In

fact, even when provided with SFL tools, developers tend not to use them and rely on traditional manual debugging approaches instead [21].

There is thus a disconnect between research and practice in the area of software debugging and, in particular, fault localization. In standard practice, a debugging task typically proceeds as follows. Developers observe that a program execution exhibits some unintended behavior, make hypotheses on what program entities caused that behavior, and confirm or reject these hypotheses by examining the execution at specific points. They then incorporate the additional knowledge acquired in the process to refine their hypotheses, possibly observing the faulty execution at different points and continuing this feedback loop until they identify the fault responsible for the observed unintended behavior. In this setting, debugging is an art that mainly relies on developers' knowledge and their familiarity with the software system being debugged.

To support the above process, while trying to automate it as much as possible, we present Swift, an iterative user-driven technique designed to help developers during debugging in a natural way. Figure 1 provides a high-level view of Swift, which performs *SFL with humans in the loop* as follows. First, it leverages traditional SFL techniques to identify and rank suspicious methods. Second, it generates queries to the developer about the correctness of specific executions of the most suspicious method. A query consists of the inputs to that method and the corresponding outputs, possibly including relevant program states. The developer answers a query by asserting whether the outputs are correct for those inputs. Third, Swift uses the response provided by the developer by incorporating it in the form of a "virtual" test case and using this additional information to improve localization results. Swift reiterates these steps until the SFL results become precise enough that the fault can be localized by the developer.

Fig. 1. Swift's interaction with the user (SUT stands for System Under Test).

A key aspect of Swift is the use of high-level abstractions to gather input from the developer. Unlike traditional SFL, in which developers are simply provided with a list of statements and must follow the list with no guidance and no additional context, Swift guides the developers towards the fault through an iterative process that only requires them to check high-level input-output relationships at the method level.

To assess the effectiveness of Swift, we implemented it in a prototype tool and performed an empirical study on 5 programs and 26 faults for these programs, while simulating the developers' answers using an automated oracle. The results of our study, albeit preliminary, are promising and provide support for further research in this direction. On average, for the faults considered, Swift required less than 10 user queries to identify the fault, which is an initial indication of the practical applicability and potential usefulness of our approach.
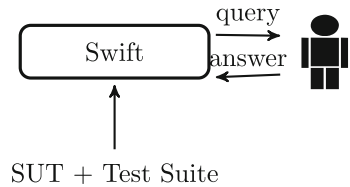
The main contributions of this paper are: (1) a novel technique that overcomes some of the limitations of existing SFL approaches by leveraging user feedback in a natural way; (2) an implementation of our approach for Java programs that is publicly available at http://www.cc.gatech.edu/~orso/software/swift/, together with our experimental infrastructure; and (3) an empirical evaluation that provides initial evidence of the potential usefulness of our approach and identifies several directions for future work.

2   Running Example

Figure 2 shows class `BoundedStack` and its test suite with two test cases, `t1` and `t2`. In this example, test `t2` fails with an `ArrayIndexOutOfBoundsException` when calling `bs.peek()`. That happens because of a fault at line 11. The method `pop()` is expected to have no effect when the stack is empty, but it incorrectly subtracts `numElems` even when `numElems == 0`, resulting in a negative stack size. A check on the size of the stack would fix the problem in this case.

For this example, SFL would rank line 18 as the most suspicious because the line is executed in the failing test case, but not in the passing test case. All other statements, except the one at line 15 have the same suspiciousness values. Thus, in this case, fault localization alone does not help the developer to diagnose the fault.

3   Approach

Figure 3 shows the workflow of Swift, which takes as input the system under test (SUT) and a test suite for the SUT with at least one failing test case.

```
1   public class BoundedStack {
2     Integer[] elems; int numElems;
3
4     BoundedStack(int max) {
5       elems = new Integer[max]; }
6
7     void push(Integer k) {
8       /* check size */
9       elems[numElems++] = k; }
10
11    void pop() { --numElems; }
12
13    Integer peek() {
14      if (size() == 0)
15        return null;
16      else return elems[size()-1];}
17
18    void clear() { numElems = 0; }
19
20    int size() { return numElems; }
21    ...
22  }
23
24  // tests
25  @Test t1() {
26    BoundedStack bs =
27      new BoundedStack(3);
28    bs.push(5); bs.push(6);
29    bs.pop();
30    assertEquals(5, bs.peek());}
31
32  @Test t2() {
33    BoundedStack bs =
34      new BoundedStack(3);
35    bs.push(7); bs.push(8);
36    bs.clear();
37    bs.pop();
38    assertEquals(null, bs.peek());
```

Fig. 2. Faulty BoundedStack [24].

In Step 1, Swift executes the provided test suite and collects runtime data about each test, including (1) coverage and pass/fail information, for performing fault localization, and (2) dynamic call information, for suitably incorporating developers' feedback. In Step 2, Swift leverages existing fault localization techniques to compute the suspiciousness of program entities based on the collected runtime information. Initially, only the executions of the existing test cases are considered. As developers interact with Swift while debugging, their knowledge regarding the examined parts of the executions is incorporated as additional runtime data, providing extra information for fault localization.

In Step 3, Swift guides the developer to examine the parts of a failing test execution where highly suspicious program entities are being executed, by means of debugging queries. In this setup, the developer is expected to check correctness of the method execution based on the provided input and output, with possibly partial state information, and give the answer back to Swift. Step 4 incorporates the developer's answer to the debugging query by modifying and augmenting the runtime data. The interactive debugging process then loops back to Step 2. Swift refines fault localization results with

Fig. 3. Workflow overview

the additional knowledge from the developer and generates another debugging query using the refined suspiciousness values. This process continues until either the fault is found or the developer gives up and stops Swift.
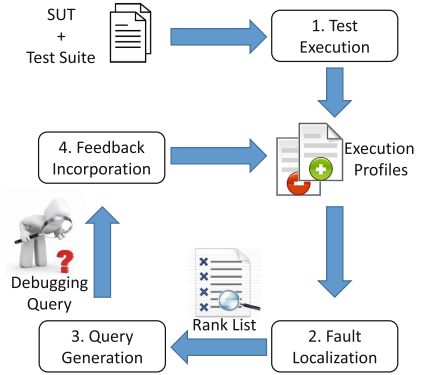
3.1   Technical Details

We now discuss the steps of Swift in detail. For each step, we first describe the step and then illustrate it on the example faulty program from Sect. 2.

Test Execution. In this initial step, Swift executes the test suite for the SUT and collects an execution tree for each test. Figure 4 shows the execution tree that corresponds to test case t2 in the BoundedStack example of Fig. 2. In the figure, each box represents a method invocation node. The labels of the nodes show the method name on the first line and the direct statement coverage information on the second line. The set of numbers inside the brackets indicates the covered statements, corresponding to the line numbers in Fig. 2.

Fault Localization. Any fault localization technique that uses coverage information to rank program entities according to their fault suspiciousness can be used in our approach. Swift currently uses Ochiai, as it has been shown to perform well in practice [3,20]. In its first iteration, Swift uses the input test suite to perform traditional fault localization. In later

Fig. 4. Execution tree corresponding to test t2 (see Fig. 2).

iterations, it also includes *virtual tests*. These additional tests model developer answers to queries in the form of synthetic execution trees and have the effect of fine tuning the fault suspiciousness values based on developer's input.
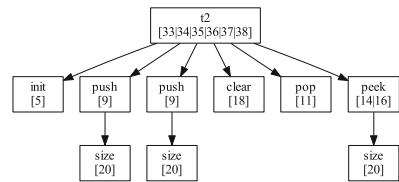
Consider the `BoundedStack` example. Assume that Swift generated a query involving method invocation `bs.push(7)` from Fig. 2 (line 35) and that the developer has examined the corresponding execution tree and determined that it is correct. This interaction produced the virtual test vt. Table 1 shows fault localization results for this scenario. Row numbers to the left of the table indicate line numbers from Fig. 2. Columns t_1, t_2, and vt show coverage for each statement in the corresponding test case. Tests t_1 and t_2 belong to the original

Table 1. Example of coverage and suspiciousness information for the `BoundedStack` example.

	✓	✗	✓	
	t_1	t_2	vt	$susp.$
5	1	1	0	0.7
9	1	1	1	0.6
11	1	1	0	0.7
14	1	1	0	0.7
15	0	0	0	0.0
16	1	1	0	0.7
18	0	1	0	1.0
20	1	1	1	0.6

test suite whereas test vt is the virtual test corresponding to the execution tree rooted at the `bs.push(7)` invocation. It reflects the developer's feedback that this method invocation produces a correct result. The symbol "✓" above the name of the test indicates that the test is passing whereas "✗" indicates a failure. Column $susp.$ shows the suspiciousness of a statement as computed by the Ochiai formula. Lines 9 and 20 have lower suspiciousness because of the additional test vt.

Query Generation. Swift asks developers for feedback through debugging queries, which basically consist of the input and output of a method invocation. Developers are expected to assess the correctness of the computation for that invocation. The rationale for this choice is that we expect the semantics of methods to be relatively well understood by developers who are familiar with the program being debugged.

Swift determines which method invocation to select for generating a debugging query from the fault localization results computed in Step 2. Swift picks, from the failing executions, a method invocation that directly covers the most suspicious statement. In case multiple statements are ranked at the top, one is chosen randomly. And if there are multiple failing executions that cover the most suspicious statement, Swift picks the invocation from the test that executes the smallest number of instances of the suspicious statement. The rationale for this heuristic is that the number of queries is a reasonable proxy for human effort, which Swift tries to minimize. Intuitively, the heuristic we presented above enables one to diagnose highly-suspicious statements more quickly, as it eagerly chooses cases that require fewer number of queries for the user to answer.

Feedback Incorporation. Developers' answers to debugging queries provide Swift with additional knowledge about the correctness of *partial program executions*. This section discusses how Swift uses this information to update the execution trees and thus incorporate developers' feedback. Figure 5 shows the pseudo-code of the algorithm for this part of our technique.

The algorithm takes as input the user feedback, represented by class `Feedback` (lines 1–5). Type `Invocation` represents a method invocation node in the

```
1    class Feedback {
2      Invocation invocation;
3      Invocation fromProfile;
4      bool isCorrect;
5    }
6
7    List<Invocation> correctProfiles = ...
8    List<Invocation> incorrectProfiles = ...
9
10   incorporateUserFeedback(Feedback feedback) {
11     if (feedback.isCorrect) {
12       feedback.invocation.removeFromParent();
13       correctProfiles.add(feedback.invocation);
14       for (Statement s
15           : feedback.invocation.getCoverage()) {
16         if (!feedback.fromProfile.covers(s)) {
17           for (Invocation incorrectProfile
18               : incorrectProfiles) {
19             incorrectProfile.removeCoverage(s); }}}
20     } else {
21       for (Invocation incorrectProfile
22           : incorrectProfiles) {
23         for (Statement s
24             : incorrectProfile.getCoverage()) {
25           if (!feedback.invocation.covers(s)) {
26             incorrectProfile.removeCoverage(s); }}}}}
```

Fig. 5. Algorithm for incorporating user feedback into execution profiles.

execution tree. Field `invocation` references the method invocation selected for this debugging query. Field `fromProfile` refers to the root of the incorrect execution profile from which `invocation` is selected. Finally, field `isCorrect` indicates developer's answer to the correctness of the invocation. Lines 7 to 8 correspond to the two sets of execution trees whose correctness is known. They are maintained by Swift throughout the debugging process and used by the fault localization component for suspiciousness computation.

For a particular debugging query, if the developer determines that the corresponding method execution is correct, Swift can conclude that this method invocation instance is not responsible for the failure (under the simplifying assumption that the developer is correct). In these situations, Swift removes the execution tree rooted at this method invocation node from its parent and marks this execution tree as a correct execution. Swift then checks whether the modified `fromProfile` no longer covers some statements it originally covered. Because these statements cannot be the cause of the failure at hand, they are excluded from consideration. Lines 12 to 19 show the pseudo-code for this case. Note that it is not necessarily true that the method invocation that is determined as correct does not exercise the faulty statement. The issue of coincidental correctness, which is considered to negatively affect the precision of fault localization techniques [4, 26], could also happen for method invocations. However, we conjecture that this issue is less likely to occur for these shorter executions.

We show an example of how positive feedback is incorporated using the `BoundedStack` program. For illustration purpose, we assume that, at the initial state, the first debugging query generated is about the method invocation `bs.push(7)` on line 35 in failing test `t2`. The developer determines that this

method invocation is correct. Figure 6a shows the execution profiles before and after this feedback is incorporated. The structure and the coverage information of the execution tree for test t1 is irrelevant and thus omitted.

A negative answer to a query corresponds to the case of a developer indicating that the method invocation in the query should have produced a different result. Swift incorporates a negative answer by reducing the set of statements potentially faulty. Specifically, Swift limits the suspicious set of statements to those that are executed by the method invocation in this debugging query by removing the coverage of all the other statements from the execution profiles (lines 21 to 26 in the pseudo-code).

Continuing the previous example on the BoundedStack program, we assume that the second debugging query generated is about the method invocation bs.pop() on line 35. This is the actual faulty method invocation, and the developer would determine that it is incorrect. Figure 6b shows the execution profiles after this negative feedback is incorporated.

3.2 Complete Debugging Session

This section applies Swift to our faulty program BoundedStack to illustrate a complete debugging session, including how Swift generates each debug query and incorporates developer feedback to update the fault localization results.

The tables in Fig. 8 show, for each iteration of Swift's debugging-query loop, the coverage matrix used for fault localization computation and the suspiciousness of each statement. Row numbers to the left of the tables correspond to the lines of code in Fig. 2. Columns t_x show the coverage of each statement in the corresponding test case. As before, tests t_1 and t_2 are in the original test suite in the code, whereas t_3 and t_4 are additional "virtual tests" created from the answers to debugging queries. Column s in each table shows the suspiciousness of the statements as computed by the Ochiai formula. Table cells with a blue background contain values that are changed with respect to the previous iteration. Due to space limit, the execution trees from which the coverage matrices are derived are not shown.

(a) Positive feedback.

(b) Negative feedback.

Fig. 6. Incorporating feedback.

The initial fault localization computation ranks line 18 at the top because it is executed only in the failing test case t_2, while the actual fault (*i.e.,* line 11) is ranked lower since it is executed in both passing and failing tests. Swift generates the first debugging query Q_1, asking the developer to examine the method invocation `bs.clear()` called on line 36. Figure 7 shows the details of the debugging query. Based on the information provided, the developer determines that this method invocation actually executed correctly. Swift removes the sub-tree that represents this method invocation from the execution tree of the failing test case t_2 and marks the sub-tree as a correct execution. The effect of this operation on the fault localization computation is shown in the second table from the left in Fig. 8. Since line 18 is executed only once in the invocation of `clear()`, its coverage flag in t_2 is removed. A "virtual test" t_3, which corresponds to the new correct execution of the `clear()` invocation, is also added to the coverage matrix. The suspiciousness of line 18 is changed to 0, as it is no longer covered by failing executions.

Q_1: `BoundedStack.clear()`#0 in t_2	
Input:	Output:
this: {	this: {
elems: {7, 8, null}	elems: {7, 8, null}
numElems: 2	numElems: 0
}	}

Fig. 7. Debugging query Q_1.

	t_1	t_2	s			t_1	t_2	t_3	s			t_1	t_2	t_3	t_4	s			t_1	t_2	t_3	t_4	s
5	1	1	0.7			1	1	0	0.7			1	1	0	0	0.7			1	0	0	0	0.0
9	1	1	0.7			1	1	0	0.7			1	1	0	1	0.6			1	0	0	1	0.0
11	1	1	0.7			1	1	0	0.7			1	1	0	0	0.7			1	1	0	0	0.7
14	1	1	0.7			1	1	0	0.7			1	1	0	0	0.7			1	0	0	0	0.0
15	0	0	0.0			0	0	0	0.0			0	0	0	0	0.0			0	0	0	0	0.0
16	1	1	0.7			1	1	0	0.7			1	1	0	0	0.7			1	0	0	0	0.0
18	0	1	1.0			0	0	1	0.0			0	0	1	0	0.0			0	0	1	0	0.0
20	1	1	0.7			1	1	0	0.7			1	1	0	1	0.6			1	0	0	1	0.0

($Q_1 \rightarrow$, $Q_2 \rightarrow$, $Q_3 \rightarrow$ transitions between tables)

Fig. 8. Complete debugging session.

After the answer to the debugging query Q_1 is incorporated, lines 5, 9, 11, 14, 16 and 20 are ranked at the top and have the same suspiciousness value. Swift randomly picks one of them to generate the next debugging query. Assume that Swift picks line 9 and selects the method invocation `bs.push(8)` called on line 35 to generate debugging query Q_2. The developer determines that the execution of this method invocation is also correct. Swift modifies the execution profiles in the same way as before. Another passing "virtual test" t_4 that covers lines 9 and 20 is added to the coverage matrix and, as a result, the suspiciousness values of these 2 lines are reduced.

For the third debugging query, Q_3, assume that Swift picks line 11, which is the actual fault. Method invocation `bs.pop()` on line 37 is selected. Figure 9 shows the details of the query. The developer determines that this method invocation is incorrect by spotting `numElems == -1` in its output data. In a realistic debugging session, the

Q_3: `BoundedStack.pop()`#0 in t_2	
Input:	Output:
this: {	this: {
elems: {7, 8, null}	elems: {7, 8, null}
numElems: 0	numElems: -1
}	}

Fig. 9. Debugging query Q_3.

developer would have found the fault at this point and concluded the debugging activity, as the faulty program state is infected in the current method invocation by a statement in its method body. For the purpose of illustration, we assume that the developer answers this debugging query instead and continues. Swift incorporates the feedback by removing the coverage flag of all the statements, except for the ones covered by the incorrect `pop()` invocation, from the failing executions. Consequently, after this debugging query is considered, the coverage of t_2 contains only line 11, which is also ranked at the top by Swift's fault localization component.

4   Empirical Evaluation

In our evaluation, we investigate the following two main research questions:

R1 – Can Swift locate the fault with a small number of debugging queries?
R2 – How does user feedback affect fault ranking?

The rest of this section describes the subject programs and faults we used (Sect. 4.1), explains the experimental setup we used (Sect. 4.2), and discusses experimental results to answer these questions (Sect. 4.3).

4.1   Subjects and Faults

To empirically assess the effectiveness of Swift, we implemented the technique in a prototype tool that works on Java programs. We evaluated the effectiveness of Swift on a benchmark with 26 faults distributed across 5 open-source applications from three repositories: SIR [2], Defects4J [16], and SAEG [1]. Each subject program contains multiple faulty versions. We selected versions that contain single non-concurrent faults that can be revealed by at least one failing test case in the original test suite. Furthermore, to better identify the benefits of Swift in the debugging process, we

Table 2. Characterization of subjects and faults.

Subject	Repo.	Fault ID	P-F	#Cls.	#Meths.	kLOC
jtopas	[2]	FAULT_2	123-3	25	251	7
		FAULT_6	125-1	25	251	7
commons-math	[2]	C_AK_1	1162-1	236	1723	43
		EDI_AK_1	1162-1	236	1723	43
		F_AK_1	1162-1	236	1723	43
		M_AK_1	1162-1	236	1723	43
		VS_AK_1	1162-1	236	1723	43
		CDI_AK_1	2048-2	477	3899	83
		MU_AK_1	2048-2	477	3899	83
		MU_AK_4	2049-1	477	3899	83
		URSU_AK_1	2048-2	477	3899	83
xml-security	[2]	CN2_AK_2	89-2	198	1278	40
		C2E_AK_1	92-2	198	1275	41
jsoup	[1]	1_3_4_b3	225-1	75	611	8
		1_4_2_b2	295-1	89	698	9
		1_5_2_b2	236-4	86	682	9
		1_5_2_b5	243-1	86	682	9
		1_6_1_b1	290-2	198	979	13
		1_6_3_b3	323-1	206	1032	14
commons-lang	[16]	b6	2125-3	169	2281	57
		b9	2057-8	170	2224	54
		b10	2055-8	170	2224	54
		b16	1913-1	160	2142	53
		b24	1698-1	143	2022	50
		b26	1677-1	139	2000	50
		b39	1566-1	123	1835	45

excluded cases where the initial fault localization results ranked the faulty statement alone at the top. The cases where the fault is initially ranked at the top together with a large number of other statements are included in the experiment because in this situation the fault is still difficult to identify and Swift can reduce the suspiciousness of non-faulty statements. Table 2 characterizes the faults we considered. Column "*Repo.*" shows the repository from which we obtained the subject. Column "*Fault ID*" shows the identifier of a given fault, as documented in their source repository. Column "*P-F*" shows the number of passing (P) and failing (F) tests for that fault. The last three columns show the number of classes, methods, and lines of code in the faulty version.

4.2   Experimental Setup

For each of the faults that we considered in the experiment, we applied Swift and recorded the number of debugging queries needed to locate the fault. We consider the fault to be located if the currently selected method invocation is the one that directly infected the state. To track how fault localization results change during the debugging process, we record the debugging query, its answer, and the updated ranking at each iteration step of Swift's main loop.

In this study, we used an automated oracle, instead of a human developer, to answer queries. For each of the faults in our experiment, we obtained the fixed version of the program and confirmed that all the tests pass. We also made sure that all code changes between the two versions involved faulty statements. Therefore, any difference in their execution must be caused by the fault. The automated oracle answers debugging queries by executing the fixed program to get the expected output of the invocation in the query, and compare it with the observed output.

One limitation of our current implementation of the automated oracle is that it does not handle infected program states in external resources (*e.g.,* files, and network communications). The subject programs we used do not have cases where the faulty state is manifested only in external resources.

There are cases in which the oracle is unable to find the corresponding method invocation that has the same input in the execution of the fixed program. This happens when the input of the method invocation in the debugging query has already been infected by the fault, and thus does not exist in the execution of the fixed version. In these cases, the oracle reports to Swift that it cannot answer the query, which is considered inconclusive and does not result in the generation of a virtual test. However, to be conservative in assessing the effectiveness of Swift, we still count these queries (*i.e.,* we add the query to the set of queries needed to locate the fault). It is important to note that, in our benchmark, these cases happen infrequently.

4.3   Results

Table 3 summarizes our results. Column "*#Queries*" shows the number of queries that Swift requires to locate the fault. Columns "*Stmt Initial Rank*"

Table 3. Summary of results.

Subject	Fault ID	#Queries	Stmt. Initial Rank	Stmt. Final Rank	Methods Initial Rank	Invocations Initial Rank
jtopas	FAULT_2	1	1/11	1/10	1/2	1/4
	FAULT_6	52	69/71	1/2	20/21	1231/1238
commons-math	C_AK_1	1	3/5	1/1	2/3	2/3
	EDI_AK_1	7	7/37	4/15	3/7	5/15
	F_AK_1	2	11/38	4/8	4/4	5/5
	M_AK_1	14	94/105	1/3	23/24	15,541/15,542
	VS_AK_1	4	2/16	2/9	1/4	1/8
	CDI_AK_1	1	12/26	11/25	2/2	2/2
	MU_AK_1	11	27/29	3/5	2/2	11/11
	MU_AK_4	11	12/36	1/6	3/6	6,065/16,971
	URSU_AK_1	1	28/37	4/13	10/10	16/16
xml-security	CN2_AK_2	1	2/9	1/2	1/1	3/5
	C2E_AK_1	26	300/456	36/67	35/49	211/348
jsoup	1_3__4_b3	5	232/248	1/1	46/52	789/795
	1_4_2_b2	8	45/49	3/7	10/10	20/20
	1_5_2_b2	10	51/60	2/8	16/18	27/29
	1_5_2_b5	1	4/20	2/4	3/6	15/25
	1_6_1_b1	19	54/59	1/6	14/14	33/33
	1_6_3_b3	4	167/176	7/14	111/112	353/359
commons-lang	b6	3	120/121	1/2	34/34	126/126
	b9	15	46/73	2/24	14/17	26/34
	b10	7	61/63	1/3	15/15	73/73
	b16	3	24/53	4/15	1/5	1/20
	b24	39	1/65	1/1	1/3	1/71
	b26	4	112/114	1/3	17/17	20/20
	b39	5	4/53	2/11	2/2	15/15

and "*Stmt Final Rank*" show the statement-level ranking of the fault before and after running Swift. Column "*Methods Initial Rank*" shows the method-level ranking of the fault in the initial state. We report rankings in the format "best-case rank/worst-case rank", as the faulty program entities can share the same suspiciousness values as other program entities. Note that we omitted the final method-level ranking. This is because the faulty methods are always ranked at the top after running Swift. It is also important to note that, in the final state of the debugging process, the statements ranked as high as (or higher than) the actual faulty statement are all in the same method that contains the fault. Column "*Invocations Initial Rank*" shows the number of method invocations to be examined before reaching the first faulty method invocation when the answers to debugging queries are not incorporated. We also refer to these numbers as the initial method-invocation ranking. These numbers are also shown in "best

case/worst case" format, depending on the position of the fault in the ranking among the statements that have the same suspiciousness value.

Overall Effectiveness. For 23 out of the 26 faulty program versions that we considered, the fault is found with less than 20 debugging queries (column "#Queries"). The average number of queries across all versions is about 10. Overall, these results indicate that a relatively small number of queries suffice to guide developers to the places where the fault infects the program state.

Note that, for this study, there are no obvious baselines to directly compare Swift against, beside a vanilla statistical fault localization approach. As the results show, inspecting code according to the output of statistical fault localization alone would be challenging (see column "Stmt. Initial Rank"). For xml-security's fault C2E_AK_1, for example, a statistical fault localization approach would require the developer to inspect 300 statements in the best case and 456 statements in the worst case. In contrast, Swift only needs 26 queries to isolate the faulty method invocation (see column "#Queries").

We note that the faulty statement is not always ranked among the most suspicious statements, even after the faulty method invocation has been isolated. This could happen if (1) executing the faulty code does not always infect the state and (2) other statements in the same method are executed more often than the number of times the state is infected. In this case, the faulty statement is covered by a relatively higher number of correct execution profiles, which reduces its suspiciousness score. Note, however, that this is not central to Swift as the technique focuses on isolating the faulty method invocation, not on optimizing the ranking of statements.

Progress of Fault Rankings. We elaborate on the effects of incorporating answers of debugging queries in Swift.

Figure 10 shows the progress of fault localization ranking as Swift incorporates answers on 2 representative faulty subject versions. We considered worst-case statement-level ranking in the plots. The x-axis denotes the number of queries answered over time and the y-axis denotes the ranking of the fault.

The plot of joup_1_4_2_b2 represents the case of the majority of the faults from the experiment (22 out of 26 cases). In this case, the ranking of the faulty statement monotonically decreases (*i.e.,* the faulty statement becomes more suspicious) from 49 down to 7
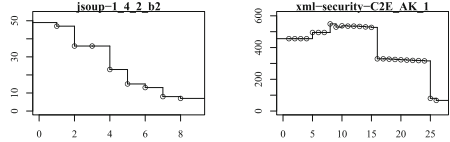
Fig. 10. Progress of stmt.-level suspiciousness.

as 8 debugging queries are answered. In contrast, the fault ranking of xml-security_C2E_AK_1 first increases, when incorporating the first 12 answers, and decreases afterwards. The reason for this type of progress pattern is that the execution of the faulty statement does not always infect the state. The first 12 queries are all classified as correct, for instance; although some of these queries

indeed cover the faulty statement, they do not infect the state. For this reason, in the beginning of the debugging process, the faulty statement appears relatively more often associated with correct execution profiles, leading to an initial increase in the ranking.

Considering all the faults we analyzed in this experiment, `xml-security_C2E_AK_1` and `commons-lang_b24` are the only ones where the fault ranking increases by a significant amount at some stage during the debugging process. We found that, for these cases, the number of queries needed to locate the fault was also larger compared to the other versions. Intuitively, these scenarios can be further explained by considering that the basic assumption of statistical fault localization is that the execution of faulty code is more correlated with failing than passing runs. However, if faulty statements infect the state infrequently when executed, this assumption becomes invalid. Swift handles these situations by calibrating suspiciousness scores of highly suspicious but non-faulty program entities.

Effect of Feedback Incorporation. In addition to looking at how Swift improves fault localization rankings, we also assessed how effective the updates to the ranking list are for guiding the search of the faulty method invocation (column "*Invocations Initial Rank*"). To that end, for each faulty version, we measured the number of queries that would be generated if Swift did not update the fault localization results using the answers to debugging queries. In this setup, Swift would start from the beginning of the initial ranking list of statements and would present all method invocations that covered the most suspicious statement to the developer. It would then go to the next statement of the ranking list when all method invocations of the current statement are answered.

By comparing this number with the number of queries needed to locate the faulty method invocation (column "*#Queries*"), we can observe a significant reduction in the number of methods to examine when query answers are considered. Furthermore, in 4 of the 26 faults, the number of method invocations to examine without feedback incorporation is larger than 500, making the task of examining all of them prohibitive. This result highlights the important role that feedback incorporation can play in the iterative debugging process.

4.4   Discussion

Although we used the ranking of statistical fault localization in our evaluation, this ranking is only used internally in Swift. The technique uses high-level abstractions to communicate results to users, who do not have to deal with the low-level abstractions used within the tool.

It is important to recall that to facilitate automation of our experiments we stop the debugging session when Swift is able to generate a query associated to a method invocation that injects the fault (see Sect. 4.2). In practice, however, users can stop (and later continue) using Swift at any point in time. This could be triggered, for instance, by the desire to check a debugging hypothesis from a suspicious fault manifestation.

The important problem of deciding how to present queries to users is outside the scope of this paper, which mainly focuses on the feasibility of the general approach. In future work, we will explore this aspect in depth and investigate different approaches, such as highlighting/obfuscating (ir)relevant fields, using program slicing, and using suitable visualization techniques. In a continuous software development environment, the users' effort of answering debugging queries can be further reduced by caching previous answers.

Our preliminary empirical evaluation shows that Swift is promising, as it provides initial evidence that Swift can locate faults by generating a relatively small number of user queries (Sect. 4.3). The evaluation also shows that our results do not seem to be coincidental, as the systematic incorporation of answers to queries improves the overall diagnosis (Sects. 4.3 and 4.3).

4.5   Threats to Validity

The main threats to validity are as follows. **External Validity**: The selection of subjects and faults we used may not generalize to other cases. To mitigate this threat, we used subjects from a variety of sources and selected according to a documented criteria, described in detail in Sect. 4.1. Another threat is that using automated oracles to simulate real users might have produced results that are not representative of a typical developer's performance. However, for an initial study that is meant to assess the feasibility of our new approach, we believe that this approximation is justified. **Internal Validity**: Errors in our implementation could affect the validity of our results. To mitigate this threat, we thoroughly checked our implementation and our experimental results, looking for discrepancies that would signal potential errors.

5   Related Work

There is an enormous body of related work on statistical fault localization and debugging in general (*e.g.,* [3, 7, 13, 14, 18, 28]). In the interest of space, and because our work builds on and extends traditional fault localization, we do not discuss this work here and focus instead on techniques that share our specific goals and general approach.

Ko and Myers proposed Whyline [17], an interactive debugger that allows developers to ask high-level questions about how values in the state came to be. Similar in spirit to dynamic backward slicing, developers can use Whyline to localize faults by iteratively asking "why" questions involving parts of the state that seem suspicious. In our approach, the tool asks questions to the developer instead, and does so by focusing on suspicious parts of the computation.

Several existing techniques use developer feedback to improve fault localization. Algorithmic Debugging (AD) [23] is a debugging technique that is popular in the functional programming community. It asks questions to testers based on the structure of the execution tree induced from one failing test and systematically prunes the tree based on the answers to get to a point where the fault can be

isolated. In contrast to Swift, AD does not take coverage profiles of multiple test runs into account to guide the debugging process. The work presented in [5,10] incorporates developer answers about the correctness of statements to refine a ranked list of suspicious statements. Swift differs from these techniques in that it asks questions about concrete input-output pairs during execution and does not rely on developers' ability to assess the correctness of individual program statements. The techniques in [11,12] suggest breakpoints using fault localization techniques and refine the suggestions based on developers' feedback on the correctness of program states. At each breakpoint, they ask the developers to examine the program states using a debugger and determine whether the state has been infected, and then increase or decrease the suspiciousness of related statements by a fixed ratio based on the feedback. In contrast, Swift generates user queries at the level of abstraction of methods in the program, whose semantics is more likely to be understood by developers than that of program states considered in isolation.

At a high-level, Swift employs a form of supervised learning to solve a program analysis problem. Recently, user supervision has been explored to solve undecidable problems in program analysis. Dillig *et al.* [9] and Mangal *et al.* [19] independently explored the feedback given by domain specialists to improve precision and recall in static analyses. In their context, user feedback indicates whether or not a warning is correct, and feedback is restricted to the output of the analysis. Swift, conversely requests user feedback on partial executions and is a dynamic (rather than static) analysis that supports debugging (rather than bug finding).

6   Conclusions

We presented Swift, a technique that aims to mitigate the existing disconnect between research and practice in the area of software debugging, and in particular in fault localization. Swift operates in an iterative and user-driven fashion. At each iteration, developers are provided with queries about the correctness of a specific method execution; Swift then processes the answers to these question and suitably increases or decreases the suspiciousness of the program entities involved in the computation. This process allows Swift to improve the localization results and guide the developer increasingly closer to the fault at hand.

We implemented Swift in a prototype that is publicly available, together with our experimental infrastructure. We used our implementation to perform an empirical evaluation of Swift on 5 programs and 26 faults for these programs. Our results show that Swift can, in most of the cases we considered, converge to the fault using only a small number of queries (less than 10, on average).

Our first goal for future work is to investigate ways to encode and visualize the queries so that they are as easy to consume and answer as possible for the developers. We will then perform a user study to assess how our approach performs in a real-world scenario, in which actual developers are answering the queries produced by Swift and performing debugging tasks.

References

1. SAEG - Software Analysis and Experimentation Group (at Universidade de São Paulo (USP), Brazil). https://github.com/saeg/experiments/tree/master/jaguar-2015
2. SIR Repository. http://sir.unl.edu/portal/index.php
3. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: An evaluation of similarity coefficients for software fault localization. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, PRDC 2006, pp. 39–46. IEEE Computer Society, Washington (2006)
4. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL, pp. 97–105 (2003)
5. Bandyopadhyay, A., Ghosh, S.: Tester feedback driven fault localization. In: Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, ICST 2012, pp. 41–50. IEEE Computer Society, Washington (2012)
6. Chandra, S., Torlak, E., Barman, S., Bodik, R.: Angelic debugging. In: ICSE, pp. 121–130 (2011)
7. Cleve, H., Zeller, A.: Locating causes of program failures. In: ICSE, pp. 342–351 (2005)
8. Demsky, B., Ernst, M.D., Guo, P.J., McCamant, S., Perkins, J.H., Rinard, M.: Inference and enforcement of data structure consistency specifications. In: ISSTA, pp. 233–244 (2006)
9. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI, pp. 181–192 (2012)
10. Gong, L., Lo, D., Jiang, L., Zhang, H.: Interactive fault localization leveraging simple user feedback. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 67–76, September 2012
11. Hao, D., Zhang, L., Xie, T., Mei, H., Sun, J.-S.: Interactive fault localization using test information. J. Comput. Sci. Technol. **24**(5), 962–974 (2009)
12. Hao, D., Zhang, L., Zhang, L., Sun, J., Mei, H.: Vida: Visual interactive debugging. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 583–586, May 2009
13. Jones, J.A., Bowring, J.F., Harrold, M.J.: Debugging in parallel. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA 2007, pp. 16–26. ACM, New York (2007)
14. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: ICSE, pp. 467–477 (2002)
15. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI, pp. 437–446 (2011)
16. Just, R., Jalali, D., Ernst, M.D.: Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), San Jose, CA, USA, pp. 437–440, July 23–25 (2014)

17. Ko, A.J., Myers, B.A.: Designing the whyline: a debugging interface for asking questions about program behavior. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2004, pp. 151–158. ACM, New York (2004)
18. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: PLDI, pp. 15–26 (2005)
19. Mangal, R., Zhang, X., Nori, A.V., Naik, M.: A user-guided approach to program analysis. In: ESEC/FSE, pp. 462–473 (2015)
20. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. **11:20**(3), 1–11:32 (2011)
21. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: ISSTA, pp. 199–209 (2011)
22. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39 (2003)
23. Shapiro, E.Y.: Algorithmic Program DeBugging. MIT Press, Cambridge (1983)
24. Stotts, P.D., Lindsey, M., Antley, A.: An informal formal method for systematic junit test case generation. In: Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002, pp. 131–143 (2002)
25. Vessey, I.: Expertise in debugging computer programs: An analysis of the content of verbal protocols. IEEE Trans. Syst. Man Cybern. B Cybern. **16**(5), 621–637 (1986)
26. Wang, X., Cheung, S.C., Chan, W.K., Zhang, Z.: Taming coincidental correctness: coverage refinement with context patterns to improve fault localization. In: Proceedings of the 31st International Conference on Software Engineering, ICSE 2009, pp. 45–55. IEEE Computer Society , Washington (2009)
27. Zhang, X., Gupta, N., Gupta, R.: Locating faults through automated predicate switching. In: ICSE, pp. 272–281 (2006)
28. Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A.: Statistical debugging: simultaneous identification of multiple bugs. In: ICML, pp. 1105–1112 (2006)