Enlightened Debugging[*]

Xiangyu Li[GT] Shaowei Zhu[GT] Marcelo d'Amorim[UFPE] Alessandro Orso[GT]

[GT]Georgia Institute of Technology
Atlanta, GA 30332-0765, USA
{xiangyu.li, swzhu, orso}@cc.gatech.edu

[UFPE]Federal Univesity of Pernambuco
Recife, PE 50740-560, Brazil
damorim@cin.ufpe.br

ABSTRACT

Numerous automated techniques have been proposed to reduce the cost of software debugging, a notoriously time-consuming and human-intensive activity. Among these techniques, Statistical Fault Localization (SFL) is particularly popular. One issue with SFL is that it is based on strong, often unrealistic assumptions on how developers behave when debugging. To address this problem, we propose Enlighten, an interactive, feedback-driven fault localization technique. Given a failing test, Enlighten (1) leverages SFL and dynamic dependence analysis to identify suspicious method invocations and corresponding data values, (2) presents the developer with a query about the most suspicious invocation expressed in terms of inputs and outputs, (3) encodes the developer feedback on the correctness of individual data values as extra program specifications, and (4) repeats these steps until the fault is found. We evaluated Enlighten in two ways. First, we applied Enlighten to 1,807 real and seeded faults in 3 open source programs using an automated oracle as a simulated user; for over 96% of these faults, Enlighten required less than 10 interactions with the simulated user to localize the fault, and a sensitivity analysis showed that the results were robust to erroneous responses. Second, we performed an actual user study on 4 faults with 24 participants and found that participants who used Enlighten performed significantly better than those not using our tool, in terms of both number of faults localized and time needed to localize the faults.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

debugging, fault localization, dynamic analysis

[*]In the title, we use the term "enlightened" with its physical, rather than spiritual, meaning of "well informed." The technique we propose is well informed because it incorporates user feedback in its otherwise automated debugging process.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180242

1 INTRODUCTION

Software debugging is a notoriously difficult, time consuming, and human-intensive activity. The task of locating the faulty code (i.e., *fault localization*), specifically, is one of the most challenging parts of debugging. For this reason, countless techniques have been proposed over the years to help developers decrease the cost of fault localization (and debugging in general). Among these approaches, one that has been particularly successful is statistical fault localization (SFL) (e.g., [6–9, 15, 16, 23, 27, 33, 34]).

The intuition behind SFL is that dynamic data collected while running passing and failing test cases can be used to perform a statistical analysis and to compute a suspiciousness value for each entity (e.g., statement, basic block, or predicate) in the program. Basically, the suspiciousness of a code entity should be directly (resp., inversely) proportional to the number of failing (resp., passing) test cases that traverse that entity. Although SFL has been a disruptive change in the area of debugging, and has generated a tremendous amount of followup research, it has some significant limitations. Most SFL techniques tend to make strong, often unrealistic assumptions on how developers behave when debugging. In particular, previous work has shown that it is unrealistic to assume that developers provided with a possibly long list of suspicious statements would go through this list in order and immediately spot the fault when they see it, without any additional context [25].

To address these limitations of SFL, while still taking advantage of its strengths, we propose Enlighten, an interactive, feedback-driven fault localization technique. We defined Enlighten so as to follow the way in which debugging is typically performed, with the goal of helping—rather than completely replacing—the humans in the loop. Typically, developers would study the failure at hand, make hypotheses on what the cause(s) of the failure may be, and examine a subset of the execution that can confirm or disprove their hypothesis. They would then leverage the additional understanding of the failure acquired in this process to make new hypotheses or refine the existing ones, examine additional subsets of the execution, and so on. This process would continue until either the developers give up or they find the fault.

Enlighten aims to mimic and support this process as follows. First, it uses traditional SFL to formulate an initial hypothesis of where the fault may be. Second, it identifies a relevant subset of the execution that can help support or negate the formulated hypothesis. Intuitively, this execution subset is identified in the form of a method invocation that results in the execution of highly suspicious entities. Third, Enlighten presents the developer with a query about the identified method invocation, expressed in terms of

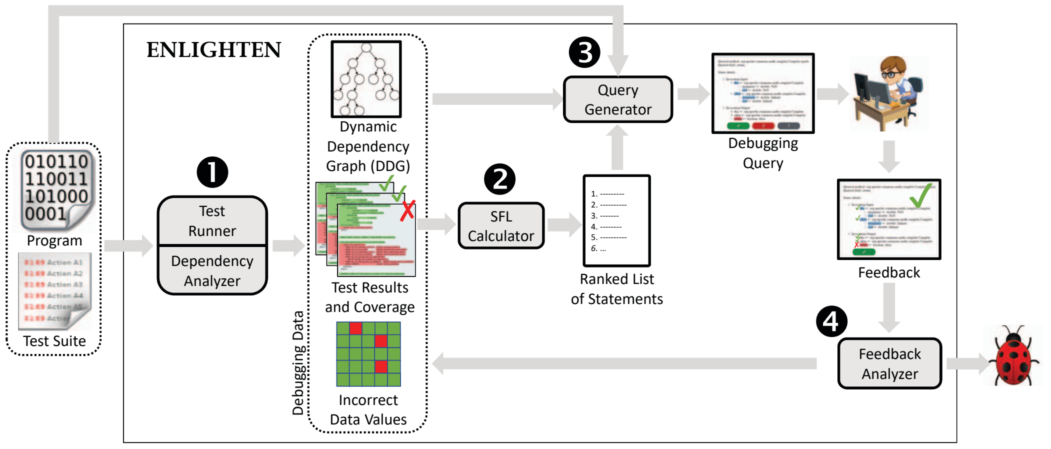

Figure 1: Overview of the approach.

the input and resulting output of the invocation. Fourth, Enlighten collects the developer feedback on the correctness of individual data values in the provided inputs and outputs and encodes this feedback as extra program specifications (i.e., extra tests that can improve the SFL results). Finally, Enlighten repeats these steps until the fault is found or the developer decides to stop.

Our approach can overcome the important limitations of traditional SFL that we highlighted earlier. Specifically, Enlighten does not require developers to decide whether a statement in isolation is correct, but rather to check high-level input-output relationships at the method level. We believe that operating at the level of abstraction of a method, whose semantics is often well understood by developers, can make the technique considerably more effective and usable. Moreover, developers can skip queries that they cannot answer and, as shown in our evaluation, the technique is resilient to occasional erroneous responses. Basically, when successful, Enlighten can nicely guide the developers towards the fault by following an iterative process that gets their input at a level of abstraction they can typically understand.

To evaluate Enlighten, we implemented the approach and performed two empirical studies. In our first study, we used an automated oracle to simulate the presence of a developer and applied Enlighten to a large number of faults in 3 open source programs. The faults we considered included 27 real faults and 1,780 mutation faults, which we generated to increase the number of data points. As our results show, for over 96% of the faults considered, Enlighten required less than 10 interactions with the simulated user to localize the fault. In the second part of our evaluation, we performed an actual user study. We selected 4 real faults and 24 participants and assigned to each participant two debugging tasks: one to be performed using Enlighten, and one to be performed using the debugging technique(s) of their choice. When using Enlighten, the participants performed considerably better than when debugging without the help of our tool. The improvement was significant in terms of both number of faults localized and time needed to localize the faults. Overall, we believe that our results provide a clear indication that Enlighten is a promising approach for debugging and fault localization.

The main contributions of this paper are:

- Enlighten, a new debugging technique that improves traditional fault localization approaches by incorporating user feedback.
- A tool that implements Enlighten and that is publicly available, together with our experiment data and infrastructure (http://www.cc.gatech.edu/~orso/software/enlighten/).
- Two complementary evaluations of Enlighten: an extensive analytical study with simulated users, in which we evaluate Enlighten on a large number of faults; and an actual user study, in which we evaluate Enlighten in a realistic debugging scenario involving real users.

2 APPROACH

Figure 1 provides a high-level view of Enlighten and shows input (left side), output (right side), and main components of the technique (inside the box). As the figure shows, Enlighten takes as input a program and its test suite and produces as output the likely location of the fault. The fault localization process of Enlighten is iterative and user-driven, as indicated by the loop and the developer's avatar in the figure. Intuitively, at the beginning of the process, Enlighten has limited knowledge about what may be causing a failure. Each iteration, however, adds relevant debugging information to Enlighten's knowledge base, which helps eventually locating the bug. In the following, we first briefly describe the main components of the technique and then discuss them in detail.

1) The *Test Runner and Dependency Analyzer* component takes as input the faulty program and a test suite for the program and computes, for each test, a dynamic dependence graph, test results, coverage information, and a set of incorrect data values.
2) The *SFL Calculator* uses the test results and the coverage information to produce a ranked list of suspicious statements, using a traditional SFL approach.
3) The *Query Generator* takes as input the program, its test suite, and the artifacts produced by the Test Runner and Dependency Analyzer, and generates debugging queries using the SFL results. Each query consists of a method invocation, together with its inputs (parameters plus relevant state) and outputs (including side effects), which the developer can mark as correct or incorrect.

4) The *Feedback Analyzer* takes as input the response to a debugging query. If the developer has found the bug, the process stops. Otherwise, the Feedback Analyzer updates the debugging data based on the developer feedback and performs another iteration.

Conceptually, Enlighten can operate with test suites that contain test cases triggering different faults. Multiple faults can negatively affect the initial SFL results. However, because Enlighten generates queries for a specific test case, the feedback provided by the user should overcome the noise introduced by the multiple faults. Moreover, there are several techniques that can cluster test cases that fail for similar reasons (e.g., [5, 14]) and that could be used to "specialize" the test suite before applying Enlighten.

It is also worth noting that debugging queries do *not* need to be formulated at the granularity of method invocations, and alternative partial program executions could be used instead. We choose to use method calls because methods are a fundamental abstraction developers use to reason about program semantics, and the behavior of many methods should be well understood by the developers.

2.1 Test Runner & Dependency Analyzer

The Test Runner is a traditional driver that takes a program and its test suite as input and produces as output the test results (pass or fail), coverage data, and a set of incorrect values. This latter is a set of incorrect values that is initialized, for each test case, with the value associated with the corresponding failure. (The Feedback Analyzer then adds to the set values marked as erroneous by the developers in response to queries.) In our implementation of Enlighten, which is for Java programs, a test failure can result in either an uncaught exception or a failing assertion. In these cases, the value associated with the failure is the reference to the object corresponding to the uncaught exception or the failed assertion, respectively.

The Dependency Analyzer, conversely, produces a Dynamic Dependence Graph (DDG) for every failing test in the test suite. In the DDG, nodes represent occurrences of statements in the program, whereas edges represent dynamic (data or control) dependences between these statements. As it is traditionally done, statements that contain more than one definition are split so that each node contains at most one definition [11].

2.2 SFL Calculator

Enlighten uses a modified version of Ochiai [3] to perform SFL. We selected Ochiai because it has been shown to perform well in practice. The specific formula we use to compute the suspiciousness of a statement s is $susp(s)=a_{ef}/\sqrt{(a_{ep}+a_{ef})\times(a_{ef}+a_{nf})}$. In the formula, a_{ef} (resp., a_{nf}) denotes the number of failing tests that covered (resp., did not cover) s. The term a_{ep} denotes the sum of the weights of the passing tests that covered s (as opposed to the number of passing tests that covered s in the original formula). The approach assigns weight 0.1 to the tests in the initial test suite and weight 1 to the virtual tests that encode the feedback provided by the user (see Section 2.4). The rationale for this decision is that we want the human feedback to have a high impact on the SFL results, as it relates to very focused partial executions. We picked these specific numbers so that there is an order of magnitude difference between the two. In future work, we plan to experiment with different weights and better understand their effect. In computing the formula, statements that are not executed in any test are assigned a suspiciousness score of 0.

2.3 Query Generator

Enlighten interacts with the developer through debugging queries, which are expressed in terms of inputs and outputs of a suspicious method invocation and are about the correctness of that invocation. The query generation process consists of (1) selecting a failing test, (2) selecting a suspicious method invocation, and (3) producing a debugging query. We now describe each of these steps.

2.3.1 Selecting a Failing Test. Enlighten generates debugging queries for a failing test. When multiple failing tests exist, and developers do not specify their test of choice, the technique selects the test that makes the smallest number of method calls. The rationale is that shorter traces should be easier to debug.

2.3.2 Selecting a Suspicious Method Invocation. Before describing how Enlighten selects suspicious method invocations, we must define the concept of *value suspiciousness*. Traditional SFL techniques (e.g., [3, 15]) associate suspiciousness scores to program statements. Enlighten uses these scores to compute the suspiciousness of values defined within a dynamic method invocation (i.e., a specific runtime instance of a method execution). In the following, `slice`$(v, invoc)$ denotes the dynamic backward slice associated with value definition v, limited to dynamic method invocation $invoc$, $v.instr$ denotes the instruction associated with definition v (i.e., the instruction that defines v), and $susp(instr)$ denotes the suspiciousness score of instruction $instr$, as computed by the SFL calculator. Enlighten computes the suspiciousness of a value definition v for a dynamic method invocation $invoc$ in two steps. First, it computes the base suspiciousness of v as $base\_susp(v, invoc) = max\{susp(v'.instr) | v' \in slice(v, invoc)\}$. It then computes the actual value suspiciousness of v ($val\_susp(v)$) based on whether v affects, through direct or indirect dependencies, values already known to be incorrect. Specifically, Enlighten computes $val\_susp(v)$ by multiplying $base\_susp(v)$ by an *amplifying factor* that is equal to either 1, if no previously labeled incorrect value depends on v, or 1 plus the number of incorrect values that depend on v, otherwise. Intuitively, values that affect others that developers previously labeled as incorrect are more suspicious.

To select the method invocation for the next query, Enlighten considers the output of all the invocations within the (failing) test execution being considered, where the output includes the state of the objects passed as parameters, the values of the modified global variables and objects, and the return value (or exception thrown).[1] For each such output item, Enlighten computes the corresponding value suspiciousness (i.e., the value suspiciousness of the corresponding definition). It then identifies the outputs with the highest value suspiciousness and selects the corresponding method invocation. In case of ties, Enlighten prioritizes methods higher in the call chain and chooses randomly when all conditions are equal.

2.3.3 Producing a Debugging Query. Conceptually, a query is a set of questions in the form "Is this value correct?". Specifically, Enlighten reports to the developer the inputs and outputs of a method

[1] Enlighten currently ignores data written through I/O operations, which could be added through additional engineering.

```
1  Set<Test> passingTests = ...
2  Set<Value> incorrectValues = ...
3
4  incorporateFeedback(Feedback feedback) {
5   if (feedback.isIncorrectInput()) {
6    incorrectValues.addAll(
7      feedback.getIncorrectInputValueDefs());
8    return;
9   }
10  for (Value v : feedback.getCorrectOutputValueDefs()) {
11    Test virtualTest = new Test();
12    virtualTest.setCoverage(slice(v, feedback.invoc));
13    passingTests.add(virtualTest);
14  }
15  if (feedback.hasIncorrectOutput()) {
16   response = askIfFaultFound();
17   if (response == ``yes'') return;
18   Set<Instr> directCov = feedback.invoc.getDirectCoverage();
19   if (response == ``no'') removeCoverage(directCov);
20   else { // ``I don't know''
21    Test virtualTest = new Test();
22    virtualTest.setCoverage(directCov);
23   }
24   incorrectValues.addAll(
25     feedback.getIncorrectOutputValueDefs());
26   Set<Instr> transitiveCov =
27     feedback.invoc.getTransitiveCoverage();
28   restrictSflTo(transitiveCov);
29 }}
```

Figure 2: Algorithm for incorporating feedback.

call that produces the most suspicious output (see Section 2.3.2) and highlights the data values with various colors (and transparency) to indicate their relative suspiciousness. Figure 4 shows an example of a debugging query where the field `numElems` is classified as highly suspicious. Developers can answer positively or negatively to any number of questions in a debugging query. A positive (resp., negative) answer indicates that the developer believes the value is correct (resp., incorrect) for that specific invocation. Intuitively, labeling an output as incorrect indicates that the bug is either in the method itself or in one of the methods it calls. Note that developers can also label an input as incorrect (e.g., an unexpected *null* value); labeling an input as incorrect tells Enlighten to ignore the current invocation and focus on methods that led to this invocation instead.

2.4 Feedback Analyzer

Figure 2 shows the algorithm for incorporating the feedback provided by developers through their answers to debugging queries (see Figure 1). Global variables `passingTests` and `incorrectValues`, declared at lines 1 and 2, store coverage information for passing tests and a set of known incorrect values observed during a debugging session. Enlighten incorporates feedback by modifying these sets. At lines 5–9, the algorithm handles the case of the developer marking some values on the input as incorrect; method `getIncorrectInputValueDefs` returns the set of value definitions specified as incorrect by the developer, and the algorithm adds those values to the set `incorrectValues` and returns.

Lines 10–14 handle the case in which the developer has labeled some output values as correct. In this case, Enlighten creates a passing virtual test for each value `v` labeled as correct and updates the debugging information accordingly: function `slice` computes the dynamic backward slice from `v`, and function `setCoverage` marks the statements in the slice as covered by the newly created virtual test. Intuitively, adding passing virtual tests reduces the suspiciousness of statements related to the computation of `v`.

Lines 15–29 handle the case in which the developer has labeled some output values as incorrect, which indicates that there may be faults in the current method or in one of the methods it calls. Enlighten therefore asks the developer to check whether the fault is in the code of the current method and to provide one of three possible answers: *yes*, *no*, *Idon′tknow* (line 16). If the developer's answer is *yes*, the fault localization process ends (line 17). If the answer is *no*, Enlighten marks all the statements in the method as not covered (by any test), which has the effect of setting to zero the suspiciousness of all instructions in this method (line 19). (Note that this does not prevent Enlighten from looking for the fault in methods called by this method.) Finally, if the answer is *Idon′tknow*, Enlighten slightly decreases the suspiciousness of the current method by adding a passing virtual test whose coverage consists of the statements directly covered by the current invocation (lines 21–22).

In these two latter cases (i.e., *no* and *Idon′tknow* answers), the fault localization process then continues. As in the case of incorrect input values, Enlighten adds output values marked as incorrect to the set of known incorrect values. Lines 26–28 then restrict the computation of SFL suspiciousness to the instructions covered, directly or indirectly, by the current invocation only.

2.5 Illustrative Example

To help illustrate the details of our approach, we introduce an example consisting of a simple faulty program. Figure 3 shows the code and corresponding test suite for class `BoundedStack`, which implements a stack of bounded size and which we adapted from previous work [30]. For brevity, we omitted the code that checks the capacity of the stack in method `push`. The fault is located at line 8: method `pop` should have no effect on an empty stack, but it does not check whether the stack is empty. Consequently, when the stack is empty, the method `pop` incorrectly decrements the field `numElems` denoting the stack size, which becomes negative. This class has three unit tests, and test `t3` fails with an `ArrayIndexOutOfBoundsException` when calling `bs.peek()` at line 35, after calling `bs.pop()` on an empty stack. At that point, the field `numElems` is -1, and the expression `size()-1` at line 12 evaluates to -2.

We now describe how Enlighten would support a developer in localizing the fault in this code.

First Iteration. The left table in Figure 5 shows the initial SFL results: line 13 is the most suspicious statement, while the actually faulty line is ranked, in the worst case, at fourth place among the eight executable statements of the program. The imprecision of SFL is caused by the fact that line 13 happens to be invoked only in the failing test case, and it thus has a stronger correlation with test failures than the actual faulty statement.

The value stored in field `numElems`, defined during the invocation of `clear`, gets its base suspiciousness score from the suspiciousness of the definition at line 13, which is 1.0. This score is then multiplied by its amplifying factor, which is computed based on the set of incorrect data values. This set initially only contains the exception object thrown when accessing the array `elems` at line 12 in `t3`. Because this exception object has a dynamic dependence on the value stored in field `numElems`, the amplifying factor associated with that value would be 2, and the value suspiciousness for `numElems` would therefore be 2.0 (see Section 2.3.2).

In this case, the value suspiciousness of `numElems` would be the highest amongst all values observed. Enlighten would therefore

```
1   public class BoundedStack {
2
3    Integer[] elems; int numElems;
4    BoundedStack(int max) { elems = new Integer[max]; }
5
6    void push(Integer k) {// check size against capacity
7     elems[numElems++] = k; }
8    void pop() { --numElems; }
9    Integer peek() {
10    if (size() = 0)
11      return null;
12    else return elems[size() - 1]; }
13   void clear() { numElems = 0; }
14   int size() { return numElems; } ... }
15
16   @Test
17   t1() {
18     BoundedStack bs = new BoundedStack(3);
19     bs.push(3);
20     assertEquals(1, bs.size()); }
21
22   @Test
23   t2() {
24     BoundedStack bs = new BoundedStack(3);
25     bs.push(4); bs.push(5);
26     bs.pop();
27     assertEquals(4, bs.peek()); }
28
29   @Test
30   t3() {
31     BoundedStack bs = new BoundedStack(3);
32     bs.push(6);
33     bs.clear();
34     bs.pop();
35     assertEquals(null, bs.peek()); }
```

Figure 3: Stack and corresponding test suite.

generate a debugging query for `clear`, shown in Figure 4, with the value of field `numElems`, marked as highly suspicious (i.e., red).

After inspecting the inputs and outputs, the developer would find that the method correctly set `numElems` to `0` and respond to the query accordingly. As a result, ENLIGHTEN would add a virtual test to the test suite reflecting the positive feedback from the developer on that output value. The coverage matrix on the right side of Figure 5 shows the updated rankings after this first iteration. Note that failing test cases and passing virtual test cases have weight 1, as described in Section 2.2.

Second Iteration. The statements at lines 8, 10, and 12 appear at the top of the ranking after the first iteration, with suspiciousness 0.95. Line 8 computes the value of `bs.numElems` in `bs.pop()`, while the execution of line 10 and 12 result in an array-out-of-bound exception in bs.peek(). The value of `bs.numElems` at the exit of `pop()` and the reference of the exception thrown by `bs.peek()` have thus a base suspiciousness of 0.95. Because the observed failure dynamically depends on both these values, their value suspiciousness is 1.90 (0.95×2). Since there are two invocations that result in the same (highest) suspiciousness value, let us assume that ENLIGHTEN randomly picks the call to function `peek` for the next query. In this case, the exception object (along with the "this" reference) would be the output of the call.

Given this query, the developer would realize that the exception is expected, as it is caused by a stack size that was already negative at the entry of the call. The developer would therefore mark field `numElems` in the input as incorrect. ENLIGHTEN would thus add the value (-1) in field `numElems` to the set of known-incorrect values, which has the effect of increasing the amplifying factor associated with all definitions that affect that value, and return (see Section 2.4).

Third Iteration. Due to the increase in its amplifying factor during the last iteration, the data value `bs.numElems` in `bs.pop()`

Test: t3		
Method: BoundedStack.clear		
Input	Output	Susp
|- elems[0]: 6	|- elems[0]: 6	0.0
|- elems[1]: 0	|- elems[1]: 0	0.0
|- elems[2]: 0	|- elems[2]: 0	0.0
|- numElems: 1	|- numElems: 0	2.0

Figure 4: Debugging query on the 1^{st} iteration.

stmt.	t1	t2	t3	*susp.*
4	1	1	1	0.91
7	1	1	1	0.91
8	0	1	1	0.95
10	0	1	1	0.95
11	0	0	0	0.00
12	0	1	1	0.95
13	0	0	1	1.00
14	1	1	1	0.91
result	✓	✓	✗	-
weight	0.1	0.1	1	-

stmt.	t1	t2	t3	t4	*susp.*
4	1	1	1	0	0.91
7	1	1	1	0	0.91
8	0	1	1	0	0.95
10	0	1	1	0	0.95
11	0	0	0	0	0.00
12	0	1	1	0	0.95
13	0	0	1	1	0.71
14	1	1	1	0	0.91
result	✓	✓	✗	✓	-
weight	0.1	0.1	1	1	-

Figure 5: Coverage matrices before / after the 1^{st} iteration.

becomes the single most suspicious value definition, with a suspiciousness score of 2.85 (0.95×3). ENLIGHTEN would therefore select the invocation of `pop()` for the third query to the developer. The developer would likely and quickly understand the failure, by observing that the value of `this.numElems` is 0 at the entry of the call and -1 at its exit, and end the fault localization process.

3 EMPIRICAL EVALUATION

We conducted two complementary studies to evaluate ENLIGHTEN: an analytical study with simulated users (Section 3.1) and a user study with real users (Section 3.2). The former let us evaluate our technique on a large number of data points and under various settings, which is typically challenging in studies involving real users. The study with real users, conversely, let us assess the usefulness of ENLIGHTEN when considering actual developers' behavior, which can only be approximated in a simulation.

3.1 Study with Simulated Users

In this study, we investigated different aspects of ENLIGHTEN using simulated users and a large number of faults. Specifically, we investigated the following research questions:

RQ1. How many iterations are necessary for ENLIGHTEN to localize a fault?

RQ2. What is the impact of the customized SFL formula and the amplifying factor on the effectiveness of ENLIGHTEN?

RQ3. How sensitive is ENLIGHTEN to incorrect user responses to debugging queries?

The first question evaluates the performance of ENLIGHTEN in a scenario in which the user always answers queries correctly. The second question assesses the usefulness of some key features of ENLIGHTEN. Finally, the third question evaluates how the performance of ENLIGHTEN degrades when the accuracy of the developers' responses degrades.

3.1.1 *Experiment Setup.*

Benchmark Programs and Faults. As benchmarks, we used three open-source programs widely used in fault localization research: Math, Lang, and Jsoup. Math and Jsoup are available in their public

Table 1: Benchmarks and faults considered.

Benchmark	*# Classes*	*# Methods*	*kLOC*	*# Faults*	
				Real	*Mutation*
Math	236 - 447	1,723 - 3,899	43 - 83	11	1,174
Lang	123 - 170	1,835 - 2,281	45 - 57	8	490
Jsoup	75 - 206	611 - 1,032	8 - 14	8	116

repositories [1], whereas Lang is available in the Defects4J repository [18]. (Math is also part of Defects4J, but with different versions from the ones we considered.) We selected these benchmarks because they do not use features unsupported by Java PathFinder [31], which Enlighten currently leverages to build DDGs. Table 1 presents these programs and faults. Since each program has multiple versions, we report the number of classes, number of methods, and code size as numeric ranges. We considered two sets of faults: (1) 27 real faults, available together with the benchmarks, and (2) 1,780 mutation faults [4, 19], which we created using the mutation tool Major [17]. We discarded faults that traditional SFL ranked in a top position, as we wanted to evaluate Enlighten in the more challenging (and more common) cases in which vanilla SFL would not be useful. This led to discarding 3 of the 30 real faults available. For the mutation faults, we ran Major in its default configuration and only considered mutants killed by at least one failing test case.

Simulated Users (Automated Oracles). We used automated oracles, in lieu of real users, to answer the queries that Enlighten generated. Consider a query involving a specific invocation i of a method. To suitably classify an output value as correct or incorrect, the oracle re-runs i using the correct version of the program and compares this expected output with that of i's actual execution. To ensure that i is executed with the same input as the faulty program, the oracle starts the test execution on the faulty version and replaces the definition of the faulty class with the correct one right before invoking i, using runtime class re-definition [2].

We assume deterministic executions, so that any difference in program state between the two runs on the faulty and correct versions can only be caused by the fault. Also, for each query, our oracle only provides feedback on the most suspicious of the output values, rather than on multiple ones. Note that providing feedback on multiple values would help locate the fault in fewer iterations, and thus likely improve the performance of our technique. However, we believe that the approach we chose mirrors well the behavior of a real user, who is more likely to focus on one or at most a few output values than on all of them. We confirmed this in our user study (see Section 3.2).

In the simulated study, Enlighten terminates when (1) the current most suspicious data value is actually faulty (i.e., it has been produced by a faulty statement), and (2) this value is computed directly in the current queried invocation. If these two conditions are not met within 100 iterations, Enlighten terminates the fault localization process and considers it failed.

Metrics. We used two metrics for evaluating the effectiveness of Enlighten: (1) the number of queries answered by the simulated user before finding the fault and (2) the number of distinct method invocations involved in such queries (the same invocation can become the most suspicious more than once). We consider these metrics reasonable approximations of developer effort: the former measures the number of interactions between the developer and the tool; the latter measures the number of times the developer needs to understand a new invocation (i.e., partial execution).

Table 2: Summary of results for real faults.

Benchmark	*Fault ID*	*IRoF*	*#Invocs*	*#Queries*		
				default	*w/o Wt*	*w/o AF*
Math	C_AK_1	5	2	2	2	4
	EDI_AK_1	37	2	2	2	6
	F_AK_1	36	3	3	4	3
	M_AK_1	112	8	10	22	13
	VS_AK_1	16	1	1	2	3
	CDI_AK_1	26	3	28	32	38
	CRVG_AK_1	62	6	23	19	19
	F_AK_2	9	1	1	1	5
	MU_AK_1	29	1	1	1	10
	MU_AK_4	36	3	3	4	8
	URSU_AK_1	13	1	1	1	1
Lang	b10	63	10	16	39	17
	b16	53	1	1	1	7
	b24	65	1	1	1	64
	b26	114	-	-	-	-
	b28	5	1	2	1	1
	b39	53	2	2	2	4
	b5	7	1	1	1	1
	b6	17	3	3	4	6
Jsoup	1_3_4-1	3	1	1	1	11
	1_3_4-3	73	4	4	4	-
	1_4_2-1	16	1	1	1	1
	1_5_2-2	21	2	2	2	2
	1_5_2-5	20	1	1	1	1
	1_6_1-1CR1	56	2	9	16	8
	1_6_1-1CR2	3	1	1	1	14
	1_6_3-3	36	5	5	3	6
Average		-	2.58	4.81	6.46	10.12

3.1.2 *RQ1: How many iterations are necessary for Enlighten to localize a fault?*

To answer RQ1, we ran Enlighten on our benchmarks and faults. We discuss the results for the real faults and those for the mutation faults separately. Table 2 shows the summary of our results for the real faults. Column "*Fault ID*" shows the identifier of the faults documented in the repositories from which we obtained them. Column "*IRoF*" (Initial Rank of Fault) shows the statement-level rank of the fault produced by SFL on the first iteration of a debugging session. Column "*# Invocs*" shows the number of distinct method invocations in the queries produced to locate the fault. Column "*# Queries*" shows the number of queries answered by the simulated user before finding the fault. Column "*default*" shows the results obtained with the default configuration of Enlighten, whereas the remaining columns show results obtained using alternative configurations (see Section 3.1.3).

Enlighten successfully localized 23 of the 27 (85%) faults within 10 iterations or less, and 26 of the 27 (96%) faults within 28 iterations or less. In 11 cases Enlighten required only 1 query to localize the fault, even though SFL did not rank the faulty line first. Considering all the faults in the study, the average number of iterations necessary for localization was 4.81 (min = 1, max = 28), and the average number of invocations involved was 2.58 (min = 1, max = 10).

We manually inspected the case of Lang b26, for which Enlighten fails to locate the fault with less than 100 queries. The faulty invocation is selected for the first time on the 15th debugging query. The suspicious output of this invocation is a string that is partially incorrect. However, due to the particular way the

Table 3: Summary of results for mutation faults.

Benchmark	*#Mut.*	*# Queries* [1, 1]		[2, 10]		[11, 100]		*Not Found*	
Math	1,174	915	77.94%	215	18.31%	29	2.47%	15	1.28%
Lang	490	438	89.39%	47	9.59%	5	1.02%	0	0.00%
Jsoup	116	77	66.38%	34	29.31%	2	1.72%	3	2.59%
Total	1,780	1,430	80.34%	296	16.63%	36	2.02%	18	1.01%

assertion of the failing test is written, the amplifying factor for all characters in the string is the same, and the oracle fails to identify the character that is actually incorrect. In subsequent debugging queries, the same faulty invocation is selected several times, but the oracle keeps missing the incorrect character for the same reason. We conjecture that in this case a real developer would be more likely to spot the error in the output string and provide the right feedback, as humans tend to view strings as a whole instead of as individual characters. (Our oracle is purposely weak to avoid unfairly favoring our technique and considers the string as multiple values, as discussed above.)

We also analyzed the correlation between *IRoF* and *# Queries* and between *IRoF* and *# Invocs*. The Pearson's correlation coefficient [26] between the number of queries and the initial rank of the fault is 0.38, which suggests a weak positive correlation. The correlation coefficient between the number of distinct invocations in the queries and the initial rank of the fault is 0.67, suggesting a moderate to strong positive correlation. Overall, the results suggest some correlation between the problem difficulty, as measured by *IRoF*, and the performance of Enlighten. However, the data also suggests that, even in cases where *IRoF* has a considerably high value, *# Queries* can be fairly low (e.g., Math.M_AK_1 and Lang.b10).

Table 3 shows the summary of our results for the mutation faults. Column "*# Queries [min, max]*" shows the number of mutants for which the number of queries needed to locate the corresponding fault was between the indicated min and max values. For example, only one query was necessary to locate 915 faults (i.e., mutants) in Math, whereas between two and ten queries were necessary to localize 47 faults in Lang. Overall, Enlighten successfully localized 99% of the 1,780 mutation faults, and on average, over 96% of all mutation faults were localized with at most 10 queries. Enlighten failed to localize the fault in only 1.01% of the cases. The results suggest that Enlighten works slightly better for mutation faults than for real faults, at least for the cases we considered. The reason may be that many of the real faults are inherently more difficult to debug—a conjecture that is potentially supported by the observation that some of these faults were present in the released versions of popular libraries.

Answering RQ1: Our results show that Enlighten can identify a large number of faults within a few iterations. With 10 or less single answers to queries, Enlighten located 85% of the real faults and 96% of the mutation faults.

3.1.3 *RQ2: What is the impact of the customized SFL formula and the amplifying factor on the effectiveness of Enlighten?*

The weights used to compute statement suspiciousness and the amplifying factor used to compute value suspiciousness are two important aspects of the design of Enlighten. This research question evaluates their effectiveness. To answer RQ2, we ran Enlighten disabling each of these features separately and compared the results so obtained with those obtained using both features.

Table 2 shows the results for this study in the columns labeled "*# Queries*". Column "*w/o Wt*" shows the number of answers to queries that Enlighten needed to locate the fault when weights were *not* taken into account (i.e., we simply set to 1 the weights of all tests, which are used to compute the term a_{ep} of the SFL formula in Section 2.2). Results show that, on average, Enlighten needed 6.46 queries in this setting, compared to 4.81 queries in the default configuration, which correspond to a 34% increase. Column "*w/o AF*" shows the number of queries when the amplifying factor (AF) was ignored (see Section 2.3.2). When using this configuration, Enlighten failed to locate the fault Jsoup.1_3_4-3 and needed, on average, 10.12 queries to locate the remaining faults. This corresponds to a 110% increase over the default configuration. Note that, due to the statistical nature of Enlighten, it is possible for the configurations "*w/o Wt*" and "*w/o AF*" to perform slightly better in some cases (e.g., Math CRVG_AK_1), but these cases are rare.

We observed similar results on mutation faults, which we do not report for space reasons. For "*w/o Wt*", the success rate of locating the fault decreased by 0.5%, and the average number of queries increased by 3.7%. For "*w/o AF*", the success rate decreased by 1.8%, and the average number of queries increased by 139%.

Answering RQ2: Results indicate that the customized SFL formula and the amplifying factor both contribute to improve Enlighten's performance. The contribution of the customized SFL formula is lower compared to the contribution of the amplifying factor.

3.1.4 *RQ3: How sensitive is Enlighten to incorrect user responses to debugging queries?*

So far, we have assumed that developers do not make mistakes. In practice, however, they can err by labeling correct values as incorrect or vice versa. This research question investigates how sensitive is the performance of Enlighten to incorrect values labeled as correct. (We leave to future work the investigation of the opposite case, which we consider less likely to happen.) To conduct this study, we modified our automated oracle so that it produced this type of erroneous answers with a configurable probability. Specifically, we considered error rates ranging from 5% to 30%, with 5 percentage points increments, and measured the number of queries and the number of cases in which Enlighten fails. As before, we configured the oracle to provide only one answer per query.

Table 4 shows, for each benchmark and for the different error rates considered, the average increase in the number of queries necessary to localize a fault over the case of an ideal oracle (i.e., a user that does not make mistakes). For example, when the erroneous answer rate is 30%, Enlighten needs, on average, 42.67% more queries to locate a fault. The results in the table show, as expected, a positive correlation between the rate of erroneous answers and the increase in the number of answers required to locate a fault. However, the results also show that Enlighten is still able to localize the fault in almost all cases. Even with 30% erroneous answers, the average success rate was higher than 99.8%.

Table 4: Sensitivity of Enlighten to human errors. Values indicate the percentual increase in the number of queries over an ideal oracle (i.e., a user that does not make mistakes).

Benchmark	Error rate					
	5%	*10%*	*15%*	*20%*	*25%*	*30%*
Math	5.92%	12.26%	18.82%	25.46%	32.08%	38.58%
Lang	6.08%	14.36%	21.93%	29.44%	36.74%	43.75%
Jsoup	7.63%	15.88%	24.37%	32.82%	41.02%	48.85%
Average	6.13%	13.89%	21.24%	28.58%	35.75%	42.67%

Answering RQ3: Although the number of queries needed to localize a fault increases with the ratio of erroneous answers, Enlighten can successfully locate the fault in most cases even in the presence of (considerable amounts of) erroneous feedback.

3.2 User Study

In addition to our study with simulated users, we conducted two actual user studies to evaluate Enlighten in a realistic scenario. Our user studies involve two debugging tasks each, where each task consists of localizing and proposing a fix for a fault in a program.

3.2.1 *Study Setup.*

Benchmarks, Faults, and Participants. The software benchmarks and faults we selected are non-trivial, real faults that existed in released versions of popular software libraries written in Java. To simulate a scenario in which participants debug code with which they are familiar, we wanted software whose semantics should be well understood by a person with a computer science background. To this end, we chose code that involves either basic mathematical concepts or XML parsing. In addition, as we did for our simulated study, we selected faults for which traditional SFL techniques do not perform well (i.e., the faulty statements are not ranked among the most suspicious statements). We do so to avoid trivial cases in which SFL by itself would be enough to localize the fault.

Table 5 summarizes the information about the two studies we performed. For each study and each task in that study, it shows the benchmark used in the task and a concise description of the corresponding fault considered. As the table shows, the faults we used for Tasks 1, 2, and 4 were selected from a benchmark used in the simulated study, whereas the fault we selected for Task 3 was used in a previous user study on SFL techniques [25]. It is worth noting that, although we used the same benchmark for Tasks 1 and 2, the parts of the program involved in the two tasks are different. In other words, completing Task 1 should not affect the participants' performance in Task 2. (Even if it had an effect, it should benefit equally participants performing Task 2 with and without Enlighten.)

Based on our assessment and observations during pilot studies, the pair of debugging tasks in each study are of similar difficulty, but the tasks in Study 2 are significantly harder than those in Study 1. This let us evaluate how Enlighten performs on faults at different difficulty levels.

For each of the studies, we recruited 12 participants (different for each study). The participants are graduate students enrolled in the computer science program either at Georgia Tech or at the Federal Univesity of Pernambuco. We also required the participants

Table 5: Debugging tasks for the user study.

User Study	*Task ID*	*Benchmark*	*Fault Description*
1	Task 1	Math	Complex number multiplication error
	Task 2	Math	Least common divisor computation error
2	Task 3	Nanoxml	XML qualified name parsing error
	Task 4	Jsoup	Absolute address construction error

to (1) have at least three years of programming experience and (2) be familiar with the Java language and the Eclipse IDE.

For each study, we randomly assigned the participants to one of two groups: Group A or Group B. Participants in Group A performed Task 1 (Study 1) or Task 3 (Study 2) without Enlighten and Task 2 (Study 1) or Task 4 (Study 2) with Enlighten. Participants in Group B performed Task 2 (Study 1) or Task 4 (Study 2) without Enlighten and Task 1 (Study 1) or Task 3 (Study 2) with Enlighten. The participants not using Enlighten were allowed to use their preferred traditional debugging approach(es) (e.g., the Eclipse IDE debugger, print statements, step-by-step execution).

We used traditional debugging approaches instead of SFL as our baseline for two reasons. First, existing studies show that SFL tends to produce no measurable advantages over traditional debugging [25, 32], so we do not expect user performance to improve using SFL instead of traditional debugging. Second, we believe that traditional debugging is a more objective baseline, as it relies on mature/well-accepted tools known to our participants.

We implemented Enlighten as a plugin for the Eclipse IDE and distributed the materials for the user study as a virtual machine image, so as to ensure a uniform experience across all participants. We informed the participants that we would measure their performance while debugging using two debugging approaches, without mentioning that Enlighten was our technique. Before the study began, the participants read a tutorial on the Enlighten plugin. When done with the tutorial, they performed their assigned debugging task. The time limit for each debugging task in Study 1 and Study 2 was 20 and 30 minutes, respectively.

In pilot studies, we found that the participants gave up on their tasks due to the complexity of the code involved and their lack of understanding of (some of) that code. Therefore, when performing the actual study, we allowed participants in all groups to ask questions about the semantics of a piece of code during the debugging process. This is akin to the common scenario in which the person who is performing the debugging task asks questions about the code to a developer with deeper knowledge of the software involved. We made sure to answer only general questions about what the methods were supposed to do, and we did not answer any questions about the faults being diagnosed.

3.2.2 *Results.*

Tables 6 and 7 show the results of the two studies. In both tables, the first two columns show the ID and the corresponding group for each participant. Columns labeled "Success" indicate whether the participant correctly identified the fault in the debugging tasks ("Y") or not ("N"). Columns labeled "Time (min)" report the time spent in localizing the fault (in case of success). For both groups, the results for the task performed using traditional debugging are shown in the 3rd and 4th columns, and the results for the task performed using Enlighten are shown in the 5th and

Table 6: Results for User Study 1.

Participant	*Group*	*Task 1 (Traditional)* *Success*	*Time (min)*	*Task 2 (Enlighten)* *Success*	*Time (min)*
1	A	Y	17.5	Y	5.4
3	A	Y	8.9	Y	11.0
5	A	Y	8.0	Y	10.8
7	A	Y	5.5	Y	8.5
9	A	Y	18.3	Y	16.0
11	A	Y	25.1	Y	16.4
		Task 2 (Traditional) *Success*	*Time (min)*	*Task 1 (Enlighten)* *Success*	*Time (min)*
2	B	Y	19.7	Y	8.6
4	B	Y	20.1	Y	9.0
6	B	Y	12.2	Y	4.0
8	B	Y	20.8	Y	9.5
10	B	Y	18.9	Y	8.4
12	B	Y	11.0	Y	5.4
Average		100%	15.5	100%	9.4

Table 7: Results for User Study 2.

Participant	*Group*	*Task 3 (Traditional)* *Success*	*Time (min)*	*Task 4 (Enlighten)* *Success*	*Time (min)*
1	A	N	-	Y	9.3
3	A	Y	21.9	Y	18.0
5	A	N	-	Y	9.6
7	A	Y	30.0	Y	5.9
9	A	Y	24.0	Y	6.0
11	A	Y	21.8	Y	9.9
		Task 4 (Traditional) *Success*	*Time (min)*	*Task 3 (Enlighten)* *Success*	*Time (min)*
2	B	N	-	Y	11.1
4	B	N	-	Y	7.4
6	B	Y	16.9	Y	7.3
8	B	Y	25.4	Y	11.0
10	B	Y	22.4	Y	4.1
12	B	N	-	Y	14.9
Average		58.3%	23.2	100%	9.5

6^{th} columns. The last row in each table shows the average success rate and debugging time for each task and technique.

In Study 1, all participants successfully completed both of their debugging tasks. On average, participants took 15.5 minutes to complete the tasks using traditional debugging, and 9.4 minutes to complete the tasks using Enlighten. Therefore, for the tasks considered, Enlighten reduced the debugging time by 39% on average. This difference is statistically significant with a p-value less than 0.005 using a one-tailed t test.

In Study 2, participants successfully completed 58.3% of the debugging tasks using traditional debugging, and the average debugging time for these successful cases was 23.2 minutes. Conversely, the participants successfully completed all their tasks when using Enlighten, and the average time spent on each task was 9.5 minutes. In these cases, therefore, the use of Enlighten increased the success rate by 71.5% and reduced the debugging time by 59%. Also in this case, the differences for both metrics are statistically significant. The p-value of the one-tailed t-test of the success rates is 0.009, and that of the debugging time is less than 0.001.

On average, for the tasks completed using Enlighten, participants needed 67% more queries than the perfect oracle to localize the faults, which indicates that humans do make mistakes in answering queries. However, it is worth noting that 71% of the participants needed exactly the same number of queries as the perfect oracle.

Comparing the reduction in debugging time in the two studies, the results seem to indicate that Enlighten improves developers' efficiency in debugging tasks more significantly for faults that are more difficult to diagnose, which we consider a positive result.

At the end of the user study, we asked the participants to complete a questionnaire about whether/how Enlighten helped them, as well as what other information could have been provided by the tool to make it easier to localize and understand the fault. The two advantages of Enlighten most frequently mentioned were that (1) it points developers to the likely faulty invocation in the execution, and (2) it provides detailed program state information for inspection. These two aspects roughly correspond to what we consider to be the main improvements we made in Enlighten over traditional debugging and traditional SFL techniques. The most wanted feature that Enlighten does not currently provide, according to the questionnaires, is the ability to get the context of the method invocation in the debugging query, including the call stack and the position of the current invocation in the entire execution. Several participants thought that this information would give them a better understanding of the entire debugging process and help them give feedback to debugging queries more efficiently. It would be straightforward to provide this additional information, and we are planning to do it in future work.

We also interviewed the participants about their general feeling on the debugging experience using Enlighten. Multiple participants mentioned that learning to use the Enlighten plugin in the time we allocated for the training was challenging. One participant specifically pointed out that it was difficult to change their debugging mindset from a traditional code-centric paradigm to a more data-centric one. Finally, several participants reported that they spent a long time inspecting the code of the method in the query only to later discover that it was not necessary. We speculate that these feedback may indicate that people's performance using Enlighten could further improve after they get more familiar with the technique.

3.3 Limitations and Threats to Validity

The main limitation of our current implementation of Enlighten comes from the computation of the dynamic dependence information. Due to the enormous engineering effort required to develop a tool that implements our approach, the current dynamic dependency analyzer does not support some features of the Java standard library (e.g., certain encryption algorithms and Swing). This is an implementation-specific limitation and can be addressed with additional engineering. Another limitation, shared with many other debugging techniques that rely on dynamic slicing, is that the performance overhead of Enlighten during program execution can be significant. In the user study, however, no participant complained about the running time of Enlighten.

The major internal threat to validity for our evaluation has to do with possible faults in our implementation of Enlighten that may invalidate our results. To address this threat, we carefully checked and unit tested our code during development. Furthermore, for the real faults in the benchmark, we manually inspected the interactions between the automated oracle and Enlighten to confirm that the sequences of debugging queries and feedback were correct.

The main external threat to validity is that the benchmarks we used might not be representative of faults in real-world scenarios and/or our results may not generalize. To mitigate this threat, we selected benchmarks that perform different types of tasks: manipulating complex data structures, performing numeric computations, and processing XML files. In addition, in the study with simulated users we evaluated Enlighten with both real faults and a large set of mutation faults, and in the study with real developers we used four different real-world faults. Another possible external threat is that the population of participants we recruited for the user study might not be representative of real developers. To mitigate this threat, we required the participants to have at least three years of programming experience.

4 RELATED WORK

Countless papers have been published on debugging and fault localization. For the sake of space, this section focuses on the work that is most closely related to Enlighten.

Algorithmic Debugging (AD) is an interactive debugging technique that was proposed in the functional programming community by Shapiro in the early eighties [28]. Similar to our technique, AD asks developers questions on the correctness of specific function invocations in the execution tree for a given failing test. The tree is then systematically pruned based on the answers to these questions until the fault can be isolated. Enlighten differs from AD in two important ways. First, AD uses basic heuristics to identify which function invocations to target, whereas Enlighten leverages SFL and dynamic dependences. Second, AD requires developers to determine whether a function invocation is completely correct, which is difficult to do in the common case of functions that involve large portions of the program state. (This problem is common to most techniques based on AD [29], including our own previous work [22], and tends to make these approaches error-prone and impractical.) Conversely, Enlighten asks developers for feedback on individual input and output values, which we believe (and our initial results show) is a more realistic approach.

Ko and Myers proposed Whyline [20, 21], an interactive debugger that lets a developer trace incorrect variable values backwards by asking questions about how these values came to be. Whyline is similar in spirit to dynamic backward slicing—the user follows a sequence of incorrect variable values through program dependence chains to get to the fault. More recently, Lin and colleagues proposed Microbat [24], a feedback-driven debugging technique that improves on Whyline by inferring patterns in execution traces and using developer feedback to skip partial program executions, expediting the backward tracing process. Enlighten, Whyline, and Microbat all leverage lightweight user feedback to improve fault localization. However, in contrast to these other techniques, the queries Enlighten produces are contextualized by method invocations as opposed to focused on arbitrary execution points. This feature not only lets the developer obtain relevant contextual information when answering specific queries, but also enables the technique to jump across calling contexts guided by the suspiciousness of program statements.

The debugging technique proposed by Hao and colleagues [12] sets breakpoints in the faulty program using suspiciousness of program statements given by SFL. At each breakpoint, the technique asks the developer to inspect the program using a debugger to determine whether the program state has been infected by the fault. The suspiciousness of related statements is then increased or decreased by a fixed ratio based on the provided feedback. In contrast to their approach, Enlighten selects for inspection a small set of suspicious data items within selected method invocations; it does not require the developers to find faulty memory locations in the entire program state. In addition, Enlighten incorporates developers' feedback into the SFL algorithm, so as to dynamically update suspiciousness information. In follow-up work, Hao and colleagues proposed VIDA [13], which leverages program dependences to find statements whose suspiciousness must be updated. Compared to VIDA, Enlighten asks for feedback on the input-output relations of methods, whose intended behavior tends to be well understood, rather than on individual program statements.

Gong and colleagues [10] proposed an interactive fault localization technique that continuously updates the ranked list of suspicious statements as the user marks statements as faulty and non-faulty. The intuition behind the technique is that, once a statement is labeled as non-faulty, the other statements executed in the same failing test case should be considered more suspicious. Like traditional SFL approaches, and unlike Enlighten, their technique requires developers to determine the correctness of individual program statements without contextual information, which has been shown to be problematic [25].

5 CONCLUSIONS

We presented Enlighten, an interactive feedback-driven fault localization technique. Enlighten combines SFL, algorithmic debugging, and dynamic dependence analysis by leveraging their strengths while mitigating their weaknesses. In particular, unlike traditional SFL, Enlighten ask developers contextualized questions that consist of queries about the inputs and outputs associated with concrete instances of suspicious method invocations. Also, unlike algorithmic debugging, Enlighten lets developers reason in terms of individual input/output data items, which is important in order to be able to handle large program states.

Our empirical results show that Enlighten is effective when applied to both real-world and mutation faults in the benchmarks we considered. Specifically, our study with simulated users shows that Enlighten can localize a majority of the faults with less than 10 debugging queries; and our user study shows that Enlighten can provide significant improvements over traditional debugging in terms of both number of faults localized and time needed to localize the faults.

In future work, we will conduct additional user studies to further investigate our core assumption that methods are a suitable level of abstraction for developers to understand program behavior during debugging. We will also perform a direct comparison between our approach and traditional SFL techniques in real-world scenarios. Finally, we will extend our implementation to incorporate it into additional IDEs and to remove some of its practical limitations.

ACKNOWLEDGMENTS

This work was partially supported by the NSF under grants CCF-1161821 and 1548856, and by CNPq under grants 457756/2014-4 and 203981/2014-6.

REFERENCES

[1] 2015. SAEG - Software Analysis and Experimentation Group (at Universidade de São Paulo (USP), Brazil). (2015). https://github.com/saeg/experiments/tree/master/jaguar-2015.

[2] 2017. Java Instrumentation API (Class Redefinition). (2017). https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC 2006)*. 39–46.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*. 402–411.

[5] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. 2009. DebugAdvisor: A Recommender System for Debugging. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE 2009)*. 373–382.

[6] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. 2003. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*. 97–105.

[7] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)*. 121–130.

[8] Holger Cleve and Andreas Zeller. 2005. Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*. 342–351.

[9] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. 2006. Inference and Enforcement of Data Structure Consistency Specifications. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA 2006)*. 233–244.

[10] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. 2012. Interactive fault localization leveraging simple user feedback. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM 2012)*. 67–76.

[11] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. 1999. An Efficient Relevant Slicing Method for Debugging. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 1999)*. 303–321.

[12] Dan Hao, Lu Zhang, Tao Xie, Hong Mei, and Jia-Su Sun. 2009. Interactive Fault Localization Using Test Information. *Journal of Computer Science and Technology* 24, 5 (2009), 962–974.

[13] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. 2009. VIDA: Visual interactive debugging. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*. 583–586.

[14] James A. Jones, James F. Bowring, and Mary Jean Harrold. 2007. Debugging in Parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA 2007)*. 16–26.

[15] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 2002)*. 467–477.

[16] Manu Jose and Rupak Majumdar. 2011. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011)*. 437–446.

[17] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. 433–436.

[18] René Just, Darioush Jalali, and Michael D. Ernst. 2014. 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 (ISSTA 2014)*. 437–440.

[19] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 654–665.

[20] Andrew J. Ko and Brad A. Myers. 2004. 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)*. 151–158.

[21] Andrew J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*. 301–310.

[22] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2016. Iterative User-Driven Fault Localization. In *Proceedings of the 12th International Haifa Verification Conference (HVC 2016)*. 82–98.

[23] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005)*. 15–26.

[24] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based Debugging. In *Proceedings of the 39th International Conference on Software Engineering (ICSE 2017)*. 393–403.

[25] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*. 199–209.

[26] Karl Pearson. 1895. Notes on regression and inheritance in case of two parents. In *Royal Society of London*. 246–263.

[27] Manos Renieris and Steven P. Reiss. 2003. Fault Localization With Nearest Neighbor Queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*. 30–39.

[28] Ehud Y. Shapiro. 1983. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA.

[29] Josep Silva. 2011. A Survey on Algorithmic Debugging Strategies. *Advances in Enginnering Software* 42, 11 (Nov. 2011), 976–991.

[30] P. David Stotts, Mark Lindsey, and Angus Antley. 2002. 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)*. 131–143.

[31] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engineering* 10, 2 (April 2003), 203–232.

[32] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. 1–11.

[33] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating Faults Through Automated Predicate Switching. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*. 272–281.

[34] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical Debugging: Simultaneous Identification of Multiple Bugs. In *Proceedings of the 23rd International Conference on Machine Learning (ICML 2006)*. 1105–1112.