Semantic Fault Localization and Suspiciousness Ranking

Maria Christakis[1(✉)], Matthias Heizmann[2(✉)], Muhammad Numair Mansur[1(✉)], Christian Schilling[3(✉)], and Valentin Wüstholz[4(✉)]

[1] MPI-SWS, Kaiserslautern, Saarbrücken, Germany
{maria,numair}@mpi-sws.org
[2] University of Freiburg, Freiburg im Breisgau, Germany
heizmann@informatik.uni-freiburg.de
[3] IST Austria, Klosterneuburg, Austria
christian.schilling@ist.ac.at
[4] ConsenSys Diligence, Berlin, Germany
valentin.wustholz@consensys.net

Abstract. Static program analyzers are increasingly effective in checking correctness properties of programs and reporting any errors found, often in the form of error traces. However, developers still spend a significant amount of time on debugging. This involves processing long error traces in an effort to localize a bug to a relatively small part of the program and to identify its cause. In this paper, we present a technique for automated fault localization that, given a program and an error trace, efficiently narrows down the cause of the error to a few statements. These statements are then ranked in terms of their suspiciousness. Our technique relies only on the semantics of the given program and does not require any test cases or user guidance. In experiments on a set of C benchmarks, we show that our technique is effective in quickly isolating the cause of error while out-performing other state-of-the-art fault-localization techniques.

1 Introduction

In recent years, program analyzers are increasingly applied to detect errors in real-world software. When detecting an error, static (or dynamic) analyzers often present the user with an error trace (or a failing test case), which shows how an assertion can be violated. Specifically, an error trace refers to a sequence of statements through the program that leads to the error. The user then needs to process the error trace, which is often long for large programs, in order to localize the problem to a manageable number of statements and identify its actual cause. Therefore, despite the effectiveness of static program analyzers in detecting errors and generating error traces, users still spend a significant amount of time on debugging.

Our Approach. To alleviate this situation, we present a technique for automated fault localization, which significantly reduces the number of statements

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 226–243, 2019.
https://doi.org/10.1007/978-3-030-17462-0_13

that might be responsible for a particular error. Our technique takes as input a program and an error trace, generated by a static analyzer, and determines which statements along the error trace are potential causes of the error. We identify the potential causes of an error by checking, for each statement along the error trace, whether there exists a local fix such that the trace verifies. We call this technique *semantic fault localization* because it exclusively relies on the semantics of the given program, without for instance requiring any test cases or guidance from the user.

Although there is existing work that also relies on program semantics for fault localization, our technique is the first to semantically rank the possible error causes in terms of suspiciousness. On a high level, we compute a *suspiciousness score* for a statement by taking into account how much code would become unreachable if we were to apply a local fix to the statement. Specifically, suspiciousness is inversely proportional to the amount of unreachable code. The key insight is that developers do not intend to write unreachable code, and thus the cause of the error is more likely to be a statement that, when fixed, renders fewer parts of the program unreachable.

Our experimental evaluation compares our technique to six fault-localization approaches from the literature on the widely-used TCAS benchmarks (of the Siemens test suite [19]). We show that in 30 out of 40 benchmarks, our technique narrows down the cause of the error more than any of the other approaches and is able to pin-point the faulty statement in 14 benchmarks. In addition, we evaluate our technique on several seeded bugs in SV-COMP benchmarks [5].

Contributions. We make the following contributions:

- We present an automated fault-localization technique that is able to quickly narrow down the error cause to a small number of suspicious statements.
- We describe an effective ranking mechanism for the suspicious statements.
- We implement this technique in a tool architecture for localizing and ranking suspicious statements along error traces reported by the Ultimate Automizer [16] software model checker.
- We evaluate the effectiveness of our technique on 51 benchmarks.

2 Guided Tour

This section uses a motivating example to give an overview of our technique for semantic fault localization and suspiciousness ranking.

Example. Let us consider the simple program on the left of Fig. 1. The statement on line 2 denotes that `y` is assigned a non-deterministic value, denoted by $\star$. The conditional on line 3 has a non-deterministic predicate, but in combination with the `assume` statements (lines 4 and 7), it is equivalent to a conditional of the following form:

```
if (y < 3) { x := 0; } else { x := 1; }
```

```
1  x := 2;
2  y := ⋆;
3  if (⋆) {
4    assume y < 3;
5    x := 0;
6  } else {
7    assume 3 ≤ y;
8    x := 1;
9  }
10 assert 0 < x;
```

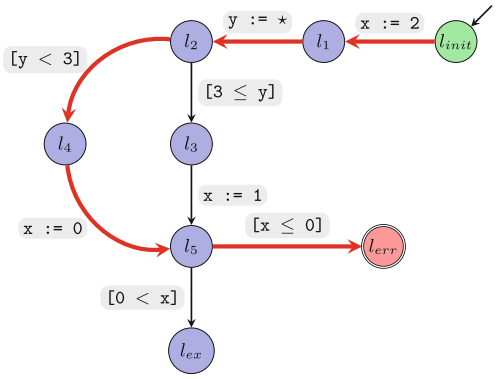

Fig. 1. The running example (left) and its control-flow graph (right).

The assertion on line 10 fails when program execution takes the `then` branch of the `if` statement, and thus, it cannot be verified by a (sound) static analyzer. The error trace that is generated by a static analyzer for this program is

`x := 2; y := ⋆; assume y < 3; x := 0; assume x ≤ 0; assert false`

and we mark it in bold on the control-flow graph of the program, which is shown on the right of Fig. 1. Statement `assume x ≤ 0` indicates that the error trace takes the failing branch of the assertion on line 10 of the program. The `assert false` statement denotes that this trace through the program results in an error.

Fault Localization. From such an error trace, our technique is able to determine a set of suspicious assignment statements, which we call *trace-aberrant statements*. Intuitively, these are statements along the error trace for which there exists a *local fix* that makes the trace verify. An assignment statement has a local fix if there exists an expression that may replace the right-hand side of the assignment such that the error becomes unreachable along this error trace. (In Sect. 5, we explain how our technique is able to identify other suspicious statements, apart from assignments.)

For example, statement `x := 0` of the error trace is trace-aberrant because there exists a value that may be assigned to variable `x` such that the trace verifies. In particular, when `x` is assigned a positive value, `assume x ≤ 0` terminates execution before reaching the error. Statement `y := ⋆` is trace-aberrant for similar reasons. These are all the trace-aberrant statements along this error trace. For instance, `x := 2` is not trace-aberrant because the value of `x` is over-written by the second assignment to the variable, thus making the error reachable along this trace, regardless of the initial value of `x`.

Suspiciousness Ranking. So far, we have seen that, when given the above error trace and the program of Fig. 1, semantic fault localization detects two trace-aberrant statements, `y := ⋆` and `x := 0`. Since for real programs there can be

many trace-aberrant statements, our technique computes a semantic suspiciousness score for each of them. Specifically, the computed score is inversely proportional to how much code would become unreachable if we applied a local fix to the statement. This is because developers do not intentionally write unreachable code. Therefore, when they make a mistake, they are more likely to fix a statement that renders fewer parts of the program unreachable relatively to other suspicious statements.

For example, if we were to fix statement `x := 0` as discussed above (that is, by assigning a positive value to `x`), no code would become unreachable. As a result, this statement is assigned the highest suspiciousness score. On the other hand, if we were to fix statement `y := ⋆` such that $3 \leq y$ holds and the trace verifies, the `then` branch of the `if` statement would become unreachable. Consequently, `y := ⋆` is assigned a lower suspiciousness score than `x := 0`.

As we show in Sect. 6, this ranking mechanism is very effective at narrowing down the cause of an error to only a few lines in the program.

Program $\mathcal{P} ::= s$

Statement $s ::= s_1; s_2 \mid v := \star \mid v := e \mid \texttt{assert}\ p \mid \texttt{assume}\ p \mid \texttt{if (}\star\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \mid \texttt{while (}\star\texttt{) \{}s\texttt{\}}$

Expression $e ::= v \mid c \mid e_1 \oplus e_2 \quad (\oplus \in \{+, -, \times\})$

Predicate $p ::= e_1 \oslash e_2 \quad (\oslash \in \{<, >, \leq, \geq, =\})$

Fig. 2. A simple programming language.

3 Semantic Fault Localization

As mentioned in the previous section, our technique consists of two steps, where we determine (trace-)aberrant statements in the first step and compute their suspiciousness ranks in the next one.

3.1 Programming Language

To precisely describe our technique, we introduce a small programming language, shown in Fig. 2. As shown in the figure, a program consists of a statement, and statements include sequencing, assignments, assertions, assumptions, conditionals, and loops. Observe that conditionals and loops have non-deterministic predicates, but note that, in combination with assumptions, they can express any conditional or loop with a predicate p. To simplify the discussion, we do not introduce additional constructs for procedure definitions and calls.

We assume that program execution terminates as soon as an assertion or assumption violation in encountered (that is, when the corresponding predicate evaluates to false). For simplicity, we also assume that our technique is applied to one failing assertion at a time.

3.2 Trace-Aberrant Statements

Recall from the previous section that trace-aberrant statements are assignments along the error trace for which there exists a local fix that makes the trace verify (that is, the error becomes unreachable along this trace):

Definition 1 (Trace aberrance). *Let τ be a feasible error trace and s an assignment statement of the form $v := \star$ or $v := e$ along τ. Statement s is* trace-aberrant *iff there exists an expression e' that may be assigned to variable v such that the trace verifies.*

To determine which assignments along an error trace are trace-aberrant, we first compute, in the post-state of each assignment, the weakest condition that ensures that the trace verifies. We, therefore, define a predicate transformer *WP* such that, if $WP(S, Q)$ holds in a state along the error trace, then the error is unreachable and Q holds after executing statement S. The definition of this weakest-precondition transformer is standard [9] for all statements that may appear in an error trace:

- $WP(s_1; s_2, Q) \equiv WP(s_1, WP(s_2, Q))$
- $WP(v := \star, Q) \equiv \forall v'.Q[v := v']$, where $v' \notin \mathit{freeVars}(Q)$
- $WP(v := e, Q) \equiv Q[v := e]$
- $WP(\texttt{assert false}, Q) \equiv \mathit{false}$
- $WP(\texttt{assume } p, Q) \equiv p \Rightarrow Q$

In the weakest precondition of the non-deterministic assignment, v' is fresh in Q and $Q[v := v']$ denotes the substitution of v by v' in Q.

To illustrate, we compute this condition in the post-state of each assignment along the error trace of Sect. 2. Weakest precondition

$$WP(\texttt{assume x} \leq \texttt{0;assert false}, \mathit{true}) \equiv 0 < \texttt{x}$$

should hold in the pre-state of statement `assume x` $\leq$ `0`, and thus in the post-state of assignment `x := 0`, for the trace to verify. Similarly, 3 $\leq$ `y` and *false* should hold in the post-state of assignments `y = ⋆` and `x := 2`, respectively. Note that condition *false* indicates that the error is always reachable after assignment `x := 2`.

Second, we compute, in the pre-state of each assignment along the error trace, the strongest condition that holds when executing the error trace until that state. We define a predicate transformer *SP* such that condition $SP(P, S)$ describes the post-state of statement S for an execution of S that starts from an initial state satisfying P. The definition of this strongest-postcondition transformer is also standard [10] for all statements that may appear in an error trace:

- $SP(P, s_1; s_2) \equiv SP(SP(P, s_1), s_2)$
- $SP(P, v := \star) \equiv \exists v'.P[v := v']$, where $v' \notin \mathit{freeVars}(P)$
- $SP(P, v := e) \equiv \exists v'.P[v := v'] \wedge v = e[v := v']$, where $v' \notin \mathit{freeVars}(P) \cup \mathit{freeVars}(e)$

- $SP(P, \texttt{assert false}) \equiv \mathit{false}$
- $SP(P, \texttt{assume } p) \equiv P \wedge p$

In the strongest postcondition of the assignment statements, v' represents the previous value of v.

For example, strongest postcondition

$$SP(\mathit{true}, \texttt{x := 2}) \equiv \texttt{x} = 2$$

holds in the post-state of assignment `x := 2`, and therefore in the pre-state of `y := ⋆`. Similarly, the strongest such conditions in the pre-state of assignments `x := 2` and `x := 0` along the error trace are *true* and $\texttt{x} = 2 \wedge \texttt{y} < 3$, respectively.

Third, our technique determines if an assignment a (of the form $v := \star$ or $v := e$) along the error trace is trace-aberrant by checking whether the Hoare triple [17] $\{\phi\}\ v := \star\ \{\psi\}$ is valid. Here, ϕ denotes the strongest postcondition in the pre-state of assignment a, v the left-hand side of a, and ψ the negation of the weakest precondition in the post-state of a. If this Hoare triple is *invalid*, then assignment statement a is trace-aberrant, otherwise it is not.

Intuitively, the validity of the Hoare triple implies that, when starting from the pre-state of a, the error is always reachable no matter which value is assigned to v. In other words, there is no local fix for statement a that would make the trace verify. Consequently, assignment a is not trace-aberrant since it cannot possibly be the cause of the error. As an example, consider statement `x := 2`. For this assignment, our technique checks the validity of the Hoare triple $\{\mathit{true}\}$ `x := ⋆` $\{\mathit{true}\}$. Since any value for `x` satisfies the true postcondition, assignment `x := 2` is not trace-aberrant.

If, however, the Hoare triple is invalid, there exists a value for variable v such that the weakest precondition in the post-state of a holds. This means that there is a local fix for a that makes the error unreachable. As a result, statement a is found to be trace-aberrant. For instance, for statement `x := 0`, we construct the following Hoare triple: $\{\texttt{x} = 2 \wedge \texttt{y} < 3\}$ `x := ⋆` $\{\texttt{x} \leq 0\}$. This Hoare triple is invalid because there are values that may be assigned to `x` such that $\texttt{x} \leq 0$ does not hold in the post-state. Assignment `x := 0` is, therefore, trace-aberrant. Similarly, for `y := ⋆`, the Hoare triple $\{\texttt{x} = 2\}$ `y := ⋆` $\{\texttt{y} < 3\}$ is invalid.

3.3 Program-Aberrant Statements

We now define *program-aberrant statements*; these are assignments for which there exists a local fix that makes *every* trace through them verify:

Definition 2 (Program aberrance). *Let τ be a feasible error trace and s an assignment statement of the form $v := \star$ or $v := e$ along τ. Statement s is* program-aberrant *iff there exists an expression e' that may be assigned to variable v such that all traces through s verify.*

Based on the above definition, the trace-aberrant assignments in the program of Fig. 1 are also program-aberrant. This is because there is only one error trace through these statements.

As another example, let us replace the assignment on line 8 of the program by `x := -1`. In this modified program, the assertion on line 10 fails when program execution takes either branch of the `if` statement. Now, assume that a static analyzer, which clearly fails to verify the assertion, generates the same error trace that we described in Sect. 2 (for the `then` branch of the `if` statement). Like before, our technique determines that statements `y := ⋆` and `x := 0` along this error trace are trace-aberrant. However, although there is still a single error trace through statement `x := 0`, there are now two error traces through `y := ⋆`, one for each branch of the conditional. We, therefore, know that `x := 0` is program-aberrant, but it is unclear whether assignment `y := ⋆` is.

To determine which trace-aberrant assignments along an error trace are also program-aberrant, one would need to check if there exists a local fix for these statements such that all traces through them verify. Recall that there exists a fix for a trace-aberrant assignment if there exists a right-hand side that satisfies the weakest precondition in the post-state of the assignment *along the error trace*. Therefore, checking the existence of a local fix for a program-aberrant statement involves computing the weakest precondition in the post-state of the statement *in the program*, which amounts to program verification and is undecidable.

Identifying which trace-aberrant statements are also program-aberrant is desirable since these are precisely the statements that can be fixed for the program to verify. However, determining these statements is difficult for the reasons indicated above. Instead, our technique uses the previously-computed weakest preconditions to decide which trace-aberrant assignments *must* also be program-aberrant, in other words, it can under-approximate the set of program-aberrant statements. In our experiments, we find that many trace-aberrant statements are *must-program-aberrant*.

To compute the must-program-aberrant statements, our technique first identifies the trace-aberrant ones, for instance, `y := ⋆` and `x := 0` for the modified program. In the following, we refer to the corresponding error trace as ϵ.

As a second step, our technique checks whether all traces through the trace-aberrant assignments verify with the most permissive local fix that makes ϵ verify. To achieve this, we instrument the faulty program as follows. We replace a trace-aberrant statement a_i of the form $v := e$ by a non-deterministic assignment $v := \star$ with the same left-hand side v. Our technique then introduces an `assume` statement right after the non-deterministic assignment. The predicate of the assumption corresponds to the weakest precondition that is computed in the post-state of the assignment along error trace ϵ[1]. We apply this instrumentation separately for each trace-aberrant statement a_i, where $i = 0, \ldots, n$, and we refer to the instrumented program that corresponds to trace-aberrant statement a_i as P_{a_i}. (Note that our technique uses this instrumentation for ranking aberrant statements in terms of suspiciousness, as we explain in Sect. 4.) Once we obtain a program P_{a_i}, we instrument it further to add a flag that allows the error to

[1] Any universal quantifier appearing in the weakest precondition can be expressed within the language of Fig. 2 by using non-deterministic assignments.

manifest itself only for traces through statement a_i (as per Definition 2). We denote each of these programs by $P^{\dagger}_{a_i}$.

For example, the light gray boxes on the right show the instrumentation for checking whether statement `x := 0` of the modified program is program-aberrant (the darker boxes should be ignored). Lines 8–9 constitute the instrumentation that generates program $P_{\texttt{x := 0}}$. As explained in Sect. 3.2, when the computed weakest precondition holds on line 9, it implies that trace ϵ verifies. Consequently, this instrumentation represents a hypothetical local fix for assignment `x := 0`. Lines 1, 10, and 15 block any program execution that does not go through statement `x := 0`. As a result, the assertion may fail only due to failing executions through this statement. Similarly, when considering the dark gray boxes in addition to lines 1 and 15 (and ignoring all other light boxes), we obtain $P^{\dagger}_{\texttt{y := }\star}$. Line 4 alone constitutes the instrumentation that generates program $P_{\texttt{y := }\star}$.

```
1  flag := 0;
2  x := 2;
3  y := ⋆;
4  assume 3 ≤ y;
5  flag := 1;
6  if (⋆)
7    assume y < 3;
8    x := ⋆;
9    assume 0 < x;
10   flag := 1;
11 } else {
12   assume 3 ≤ y;
13   x := -1;
14 }
15 assume flag = 1;
16 assert 0 < x;
```

Third, our technique runs the static analyzer on each of the n instrumented programs $P^{\dagger}_{a_i}$. If the analyzer does not generate a new error trace, then statement a_i must be program-aberrant, otherwise we do not know. For instance, when running the static analyzer on $P^{\dagger}_{\texttt{x := 0}}$ from above, no error is detected. Statement `x := 0` is, therefore, program-aberrant. However, an error trace is reported for program $P^{\dagger}_{\texttt{y := }\star}$ (through the `else` branch of the conditional). As a result, our technique cannot determine whether `y := ⋆` is program-aberrant. Notice, however, that this statement is, in fact, not program-aberrant because there is no fix that we can apply to it such that both traces verify.

3.4 *k*-Aberrance

So far, we have focused on (trace- or program-) aberrant statements that may be fixed to single-handedly make one or more error traces verify. The notion of aberrance, however, may be generalized to sets of statements that make the corresponding error traces verify only when fixed together:

Definition 3 (k-Trace aberrance). *Let τ be a feasible error trace and $\bar{s}$ a set of assignment statements of the form $v := \star$ or $v := e$ along τ. Statements $\bar{s}$ are $|\bar{s}|$-*trace-aberrant*, where $|\bar{s}|$ is the cardinality of $\bar{s}$, iff there exist local fixes for all statements in $\bar{s}$ such that trace τ verifies.*

Definition 4 (k-Program aberrance). *Let $\bar{\tau}$ be the set of all feasible error traces through any assignment statement s in a set $\bar{s}$. Each statement s is of the form $v := \star$ or $v := e$ along an error trace τ in $\bar{\tau}$. Statements $\bar{s}$ are $|\bar{s}|$-*program-aberrant*, where $|\bar{s}|$ is the cardinality of $\bar{s}$, iff there exist local fixes for all statements in $\bar{s}$ such that all traces $\bar{\tau}$ verify.*

For example, consider the modified version of the program in Fig. 1 that we discussed above. Assignments `x := 0` and `x := -1` are 2-program-aberrant because their right-hand side may be replaced by a positive value such that both traces through these statements verify.

Our technique may be adjusted to compute k-aberrant statements by exploring all combinations of k assignments along one or more error traces.

4 Semantic Suspiciousness Ranking

In this section, we present how aberrant statements are ranked in terms of their suspiciousness. As mentioned earlier, the suspiciousness score of an aberrant statement is inversely proportional to how much code would become unreachable if we applied a local fix to the statement.

First, for each aberrant statement a_i, where $i = 0, \ldots, n$, our technique generates the instrumented program P_{a_i} (see Sect. 3.3 for the details). Recall that the trace-aberrant statements for the program of Fig. 1 are `y := ⋆` and `x := 0`.

Second, we check reachability of the code in each of these n instrumented programs P_{a_i}. Reachability may be simply checked by converting all existing assertions into assumptions and introducing an `assert false` at various locations in the program. An instrumentation for checking reachability in $P_{\texttt{x := 0}}$ is shown on the right; all changes are highlighted. In particular, we detect whether the injected assertion on line 7 is reachable by passing the above program (without the dark gray box) to an off-the-shelf analyzer. We can similarly check reachability of the other assertion. In the above program, both assertions are reachable, whereas in the corresponding program for assignment `y := ⋆`, only one of them is. The number of reachable assertions in a program P_{a_i} constitutes the suspiciousness score of statement a_i.

```
1  x := 2;
2  y := ⋆;
3  if (⋆)
4    assume y < 3;
5    x := ⋆;
6    assume 0 < x;
7    assert false;
8  } else {
9    assume 3 ≤ y;
10   x := 1;
11   assert false;
12 }
13 assume 0 < x;
```

As a final step, our technique ranks the aberrant statements in order of decreasing suspiciousness. Intuitively, this means that, by applying a local fix to the higher-ranked statements, less code would become unreachable in comparison to the statements that are ranked lower. Since developers do not typically intend to write unreachable code, the cause of the error in P is more likely to be a higher-ranked aberrant statement. For our running example, trace-aberrant statement `x := 0` is ranked higher than `y := ⋆`.

As previously discussed, when modifying the program of Fig. 1 to replace assignment `x := 1` by `x := -1`, our technique determines that only `x := 0` must be program-aberrant. For the error trace through the other branch of the conditional, we would similarly identify statement `x := -1` as must-program-aberrant. Note that, for this example, must program aberrance does not miss any program-aberrant statements. In fact, in our experiments, must program aberrance does not miss any error causes, despite its under-approximation.

5 Implementation

We have implemented our technique in a toolchain for localizing and ranking suspicious statements in C programs. We used UAutomizer in the Ultimate analysis framework to obtain error traces (version `0.1.23`). UAutomizer is a software model checker that translates C programs to Boogie [4] and then employs an automata-based verification approach [16]. Our implementation extends UAutomizer to identify (trace- or program-) aberrant statements along the generated error traces, as we describe in Sect. 3. Note that, due to abstraction (for instance, of library calls), UAutomizer may generate spurious error traces. This is an orthogonal issue that we do not address in this work.

To also identify aberrant expressions, for instance, predicates of conditionals or call arguments, we pre-process the program by first assigning these expressions to temporary variables, which are then used instead. This allows us to detect error causes relating to statements other than assignments.

Once the aberrant statements have been determined, we instrument the Boogie code to rank them (see Sect. 4). Specifically, our implementation inlines procedures and injects an `assert false` statement at the end of each basic block (one at a time). Instead of extending the existing support for "smoke checking" in Boogie, we implemented our own reachability checker in order to have more control over where the assertions are injected. While this might not be as efficient due to the larger number of Boogie queries (each including the time for parsing, pre-processing, and SMT solving), one could easily optimize or replace this component.

6 Experimental Evaluation

We evaluate the effectiveness of our technique in localizing and ranking suspicious statements by applying our toolchain to several faulty C programs. In the following, we introduce our set of benchmarks (Sect. 6.1), present the experimental setup (Sect. 6.2), and investigate four research questions (Sect. 6.3).

6.1 Benchmark Selection

For our evaluation, we used 51 faulty C programs from two independent sources. On the one hand, we used the faulty versions of the TCAS task from the Siemens test suite [19]. The authors of the test suite manually introduced faults in several tasks while aiming to make these bugs as realistic as possible. In general, the Siemens test suite is widely used in the literature (e.g., [14,20,22,25–28]) for evaluating and comparing fault-localization techniques.

The TCAS task implements an aircraft-collision avoidance system and consists of 173 lines of C code; there are no specifications. This task also comes with 1608 test cases, which we used to introduce assertions in the faulty program versions. In particular, in each faulty version, we specified the correct behavior as this was observed by running the tests against the original, correct version of the

code. This methodology is commonly used in empirical studies with the Siemens test suite, and it was necessary for obtaining an error trace from UAutomizer.

On the other hand, we randomly selected 4 correct programs (with over 250 lines of code) from the SV-COMP software-verification competition [5], which includes standard benchmarks for evaluating program analyzers. We automatically injected faults in each of these programs by randomly mutating statements within the program. All SV-COMP benchmarks are already annotated with assertions, so faults manifest themselves by violating the existing assertions.

6.2 Experimental Setup

We ran all experiments on an Intel® Core i7 CPU @ 2.67 GHz machine with 16 GB of memory, running Linux. Per analyzed program, we imposed a timeout of 120 s and a memory limit of 6 GB to UAutomizer.

To inject faults in the SV-COMP benchmarks, we developed a mutator that randomly selects an assignment statement, mutates the right-hand side, and checks whether the assertion in the program is violated. If it is, the mutator emits a faulty program version. Otherwise, it generates up to two additional mutations for the same assignment before moving on to another.

6.3 Experimental Results

To evaluate our technique, we consider the following research questions:

- **RQ1:** How effective is our technique in narrowing down the cause of an error to a small number of suspicious statements?
- **RQ2:** How efficient is our technique?
- **RQ3:** How does under-approximating program-aberrant statements affect fault localization?
- **RQ4:** How does our technique compare against state-of-the-art approaches for fault localization in terms of effectiveness and efficiency?

RQ1 (Effectiveness). Tables 1 and 2 summarize our experimental results on the TCAS and SV-COMP benchmarks, respectively. The first column of Table 1 shows the faulty versions of the program, and the second column the number of trace-aberrant statements that were detected for every version. Similarly, in Table 2, the first column shows the program version[2], the second column the lines of source code in every version, and the third column the number of trace-aberrant statements. For all benchmarks, the actual cause of each error is always

[2] A version is denoted by <*correct-program-id*>.<*faulty-version-id*>. We mutate the following correct programs from SV-COMP: 1. `mem_slave_tlm.1_true-unreach-call_false-termination.cil.c` (4 faulty versions), 2. `kundu_true-unreach-call_false-termination.cil.c` (4 faulty versions), 3. `mem_slave_tlm.2_true-unreach-call_false-termination.cil.c` (2 faulty versions), and 4. `pc_sfifo_1_true-unreach-call_false-termination.cil.c` (1 faulty version). All versions are at: https://github.com/numairmansur/SemanticFaultLocalization_Benchmarks.

Table 1. Our experimental results for the TCAS benchmarks.

Prg	Abr stmts		Rank		Time (m:s)		Program reduction (%)							
ver	trc	prg	trc	prg	trc	prg	trc	prg	A	B	C	D	E	F
1	16	11	5	3	2:24	1:17	8.1	5.7	1.7	1.7	1.7	1.7	1.7	8.6
2	15	13	1	1	2:33	1:46	1.7	1.7	6.3	5.7	5.7	6.3	5.7	4.6
3	4	4	1	1	0:29	0:31	1.7	1.7	12.7	12.1	10.4	12.7	12.1	9.8
4	16	14	4	4	2:59	1:53	7.5	7.5	1.7	1.7	1.7	1.7	1.7	9.2
5	4	4	1	1	0:28	0:33	1.7	1.7	14.4	10.9	9.8	14.4	10.9	8.6
6	13	13	5	5	1:51	1:53	7.5	7.5	2.8	2.8	2.8	2.8	2.8	8.6
7	15	13	5	5	2:28	1:57	6.9	6.3	12.1	12.1	9.8	12.1	12.1	9.2
8	14	13	5	5	2:14	1:51	6.9	5.7	12.7	12.7	12.7	12.7	12.7	8.6
9	16	14	1	1	2:21	1:55	1.7	1.7	7.5	7.5	7.5	7.5	7.5	5.2
10	15	14	2	2	2:12	1:53	3.4	2.8	11.5	11.5	15.0	11.5	16.1	9.2
11	15	7	2	1	1:58	0:56	2.8	1.7	2.8	2.8	2.3	2.8	2.8	6.3
12	16	14	3	3	2:38	1:46	5.7	5.2	12.7	12.1	9.8	12.7	12.1	9.2
13	17	15	5	5	2:40	1:56	6.3	5.2	15.6	12.1	10.4	15.6	12.1	9.2
14	4	4	1	1	0:30	0:29	1.7	1.7	5.7	5.7	5.7	5.7	5.7	8.1
15	17	15	4	4	2:52	2:02	7.5	6.3	15.0	13.2	10.4	15.0	13.2	7.5
16	16	14	5	5	2:22	1:57	6.3	6.3	12.1	12.1	12.1	12.1	12.1	9.2
17	16	14	5	5	2:37	2:03	8.0	7.5	12.1	12.1	9.8	12.1	12.1	9.2
18	16	14	5	5	2:02	1:51	8.0	7.5	14.4	11.5	9.8	14.4	11.5	6.9
19	16	14	5	5	2:20	1:55	8.0	7.5	12.1	12.1	9.8	12.1	12.1	9.2
20	15	11	1	1	2:16	1:23	1.7	1.7	7.5	7.5	7.5	7.5	7.5	9.2
21	16	12	1	1	2:16	1:27	1.7	1.7	7.5	7.5	7.5	7.5	7.5	8.6
22	17	8	1	1	2:45	1:03	2.3	1.7	7.5	7.5	7.5	7.5	7.5	5.7
23	14	11	1	1	2:22	1:30	1.7	1.7	7.5	7.5	7.5	7.5	7.5	6.3
24	17	13	1	1	2:39	1:57	2.3	1.7	7.5	7.5	7.5	7.5	7.5	8.6
25	16	14	3	3	1:44	1:51	9.2	8.1	1.1	1.1	1.1	1.1	1.1	6.9
26	15	13	3	3	2:17	1:52	2.8	1.7	13.8	12.7	10.4	13.8	12.7	9.2
27	15	13	4	4	2:36	1:58	8.6	8.1	14.4	10.9	9.8	14.4	10.9	10.9
28	15	13	1	1	2:03	1:50	1.7	1.7	6.3	5.7	4.0	6.3	5.7	5.7
29	12	8	3	2	1:46	1:09	1.7	1.7	6.3	6.3	5.7	6.3	5.7	5.7
30	14	8	2	2	1:47	1:06	1.7	1.7	6.3	5.7	5.7	6.3	5.7	5.7
31	17	15	4	4	2:32	1:57	8.1	6.3	2.8	2.8	2.8	2.8	2.8	10.9
32	14	13	5	5	1:55	1:54	6.3	6.3	2.8	2.8	2.8	2.8	2.8	10.9
33	17	15	4	4	2:48	2:05	2.8	2.8	14.4	12.7	10.9	14.4	12.7	—
34	16	14	4	4	2:43	2:13	8.0	6.9	13.2	12.7	10.4	13.2	12.7	8.6
35	14	8	2	2	1:59	0:59	2.3	1.7	6.3	5.7	4.0	6.3	5.7	5.7
36	6	3	1	1	0:43	0:26	3.4	1.7	13.2	13.2	10.4	13.2	13.2	2.9
37	17	8	6	3	2:43	1:04	9.8	4.6	1.1	1.1	0.5	1.1	0.5	8.6
39	16	7	5	1	2:45	0:55	9.2	4.6	1.1	1.1	1.1	1.1	1.1	6.9
40	16	14	4	4	2:48	1:46	6.3	6.3	15.0	15.0	15.0	15.0	15.6	6.3
41	17	15	4	4	2:50	2:00	9.8	8.6	1.7	1.7	1.7	1.7	1.7	8.6
Avg	14.4	11.5	3.1	2.9	2:12	1:34	5.1	4.3	8.6	8.0	7.3	8.6	8.1	7.9

included in the statements that our technique identifies as trace-aberrant. This is to be expected since the weakest-precondition and strongest-postcondition transformers that we use for determining trace aberrance are known to be sound.

The fourth column of Table 1 and the fifth column of Table 2 show the suspiciousness rank that our technique assigns to the actual cause of each error. For both sets of benchmarks, the average rank of the faulty statement is 3, and all faulty statements are ranked in the top 6. A suspiciousness rank of 3 means that

users need to examine *at least* three statements to identify the problem; they might have to consider more in case multiple statements have the same rank.

To provide a better indication of how much code users have to examine to identify the bug, the eighth column of Table 1 and the seventh column of Table 2 show the percentage reduction in the program size. On average, our technique reduces the code size down to 5% for TCAS and less than 1% for SV-COMP.

RQ2 (Efficiency). The sixth column of Table 1 shows the time that our technique requires for identifying the trace-aberrant statements in a given error trace as well as for ranking them in terms of suspiciousness. This time does not include the generation of the error trace by UAutomizer. As shown in the table, our technique takes only a little over 2 min on average to reduce a faulty program to about 5% of its original size.

Table 2. Our experimental results for the SV-COMP benchmarks.

Prg ver	LoSC	Abr stmts		Rank		Rdc (%)	
		trc	prg	trc	prg	trc	prg
1.1	1336	34	29	4	2	0.4	0.2
1.2	1336	34	28	4	2	0.4	0.2
1.3	1336	34	31	2	1	0.2	0.1
1.4	1336	34	30	3	1	0.4	0.1
2.1	630	23	10	3	2	1.3	0.3
2.2	630	16	8	1	1	0.3	0.3
2.3	630	22	9	3	2	1.3	0.3
2.4	630	27	25	4	3	1.2	1.1
3.1	1371	37	33	3	1	0.3	0.2
3.2	1371	37	32	3	1	0.3	0.1
4.1	360	18	8	4	2	3.3	0.8
Average		28.7	22.0	3.0	1.6	0.8	0.3

Note that most of this time (98.5% on average) is spent on the suspiciousness ranking. The average time for determining the trace-aberrant statements in an error trace is only 1.7 s. Recall from Sect. 5 that our reachability analysis, which is responsible for computing the suspiciousness score of each aberrant statement, is not implemented as efficiently as possible (see Sect. 5 for possible improvements).

RQ3 (Program aberrance). In Sect. 3.3, we discussed that our technique can under-approximate the set of program-aberrant statements along an error trace. The third, fifth, seventh, and ninth columns of Table 1 as well as the fourth, sixth, and eighth columns of Table 2 show the effect of this under-approximation.

There are several observations to be made here, especially in comparison to the experimental results for trace aberrance. First, there are fewer aberrant statements, which is to be expected since (must-)program-aberrant statements may only be a subset of trace-aberrant statements. Perhaps a bit surprisingly, the actual cause of each error is always included in the must-program-aberrant statements. In other words, the under-approximation of program-aberrant statements does not miss any error causes in our benchmarks. Second, the suspiciousness rank assigned to the actual cause of each error is slightly higher, and all faulty

statements are ranked in the top 5. Third, our technique requires about 1.5 min for fault localization and ranking, which is faster due to the smaller number of aberrant statements. Fourth, the code is reduced even more, down to 4.3 for TCAS and 0.3% for SV-COMP.

RQ4 (Comparison). To compare our technique against state-of-the-art fault-localization approaches, we evaluated how five of the most popular [35] spectrum-based fault-localization (SBFL) techniques [20,24,34] perform on our benchmarks. In general, SBFL is the most well-studied and evaluated fault-localization technique in the literature [26]. SBFL techniques essentially compute suspiciousness scores based on statement-execution frequencies. Specifically, the more frequently a statement is executed by failing test cases and the less frequently it is executed by successful tests, the higher its suspiciousness score. We also compare against an approach that reduces fault localization to the maximal satisfiability problem (MAX-SAT) and performs similarly to SBFL.

The last eight columns of Table 1 show the comparison in code reduction across different fault-localization techniques. Columns A, B, C, D, and E refer to the SBFL techniques, and in particular, to Tarantula [20], Ochiai [2], Op2 [24], Barinel [1], and DStar [34], respectively. The last column (F) corresponds to BugAssist [21,22], which uses MAX-SAT. To obtain these results, we implemented all SBFL techniques and evaluated them on TCAS using the existing test suite. For BugAssist, we used the published percentages of code reduction for these benchmarks [22]. Note that we omit version 38 in Table 1 as is common in experiments with TCAS. The fault is in a non-executable statement (array declaration) and its frequency cannot be computed by SBFL.

The dark gray boxes in the table show which technique is most effective with respect to code reduction for each version. Our technique for must program aberrance is the most effective for 30 out of 40 versions. The light gray boxes in the trace-aberrance column denote when this technique is the most effective in comparison with columns A–F (that is, without considering program aberrance). As shown in the table, our technique for trace aberrance outperforms approaches A–F in 28 out of 40 versions. In terms of lines of code, users need to inspect 7–9 statements when using our technique, whereas they would need to look at 13–15 statements when using other approaches. This is a reduction of 4–8 statements, and every statement that users may safely ignore saves them valuable time.

Regarding efficiency, our technique is comparable to SBFL (A–E); we were not able to run BugAssist (F), but it should be very lightweight for TCAS. SBFL techniques need to run the test suite for every faulty program. For the TCAS tests, this takes 1 min 11 s on average on our machine. Parsing the statement-execution frequencies and computing the suspiciousness scores takes about 5 more seconds. Therefore, the average difference with our technique ranges from a few seconds (for program aberrance) to a little less than a minute (for trace aberrance). There is definitely room for improving the efficiency of our technique, but despite it being slightly slower than SBFL for these benchmarks, it saves the user the effort of inspecting non-suspicious statements. Moreover, note that the larger the test suite, the higher the effectiveness of SBFL, and the

longer its running time. Thus, to be as effective as our technique, SBFL would require more test cases, and the test suite would take longer to run. We do not consider the time for test case generation just like we do not consider the running time of the static analysis that generates the error traces.

7 Related Work

Among the many fault-localization techniques [35], SBFL [20,24,34] is the most well-studied and evaluated. Mutation-based fault localization (MBFL) [23,25] is almost as effective as SBFL but significantly more inefficient [26]. In general, MBFL extends SBFL by considering, not only how frequently a statement is executed in tests, but also whether a mutation to the statement affects the test outcomes. So, MBFL generates many mutants per statement, which requires running the test suite per mutant, and not per faulty program as in SBFL. Our local fixes resemble mutations, but they are performed *symbolically* and can be seen as applying program-level abductive reasoning [6,11,12] or angelic verification [8] for fault localization.

The use of error invariants [7,13,18,29] is a closely-related fault-localization technique. Error invariants are computed from Craig interpolants along an error trace and capture which states will produce the error from that point on. They are used for slicing traces by only preserving statements whose error invariants before and after the statement differ. Similarly, Wang et al. [32] use a syntactic-level weakest-precondition computation for a given error trace to produce a minimal set of word-level predicates, which explain why the program fails. In contrast, we use the novel notion of trace aberrance for this purpose and compute a suspiciousness ranking to narrow down the error cause further.

Griesmayer et al. [14] use an error trace from a bounded model checker to instrument the program with "abnormal predicates". These predicates allow expressions in the program to take arbitrary values, similarly to how our technique replaces a statement $v := e$ by a non-deterministic one. Unlike our technique, their approach may generate a prohibitively large instrumentation, requires multiple calls to the model checker, and does not rank suspicious statements.

Several fault-localization algorithms leverage the differences between faulty and successful traces [3,15,27,36]. For instance, Ball et al. [3] make several calls to a model checker and compare any generated counterexamples with successful traces. In contrast, we do not require successful traces for comparisons.

Zeller [36] uses delta-debugging, which identifies suspicious parts of the input by running the program multiple times. Slicing [31,33] removes statements that are definitely not responsible for the error based on data and control dependencies. Shen et al. [30] use unsatisfiable cores for minimizing counterexamples. Our technique is generally orthogonal to these approaches, which could be run as a pre-processing step to reduce the search space.

8 Conclusion

We have presented a novel technique for fault localization and suspiciousness ranking of statements along an error trace. We demonstrated its effectiveness in narrowing down the error cause to a small fraction of the entire program.

As future work, we plan to evaluate the need for k-aberrance by analyzing software patches and to combine our technique with existing approaches for program repair to improve their effectiveness.

Acknowledgments. This work was supported by the German Research Foundation (DFG) as part of CRC 248 (https://www.perspicuous-computing.science), the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award), and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 754411.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In: ASE, pp. 88–99. IEEE Computer Society (2009)
2. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C.: A practical evaluation of spectrum-based fault localization. JSS **82**, 1780–1792 (2009)
3. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL, pp. 97–105. ACM (2003)
4. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
5. Beyer, D.: Competition on software verification (SV-COMP) (2017). https://sv-comp.sosy-lab.org
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300. ACM (2009)
7. Christ, J., Ermis, E., Schäf, M., Wies, T.: Flow-sensitive fault localization. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 189–208. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_13
8. Das, A., Lahiri, S.K., Lal, A., Li, Y.: Angelic verification: precise verification modulo unknowns. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 324–342. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_19
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. CACM **18**, 453–457 (1975)
10. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, New York (1990). https://doi.org/10.1007/978-1-4612-3228-5
11. Dillig, I., Dillig, T.: Explain: a tool for performing abductive inference. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 684–689. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_46
12. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI, pp. 181–192. ACM (2012)

13. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 187–201. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_17
14. Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for C programs. ENTCS **174**, 95–111 (2007)
15. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. STTT **8**, 229–247 (2006)
16. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_2
17. Hoare, C.A.R.: An axiomatic basis for computer programming. CACM **12**, 576–580 (1969)
18. Holzer, A., Schwartz-Narbonne, D., Tabaei Befrouei, M., Weissenbacher, G., Wies, T.: Error invariants for concurrent traces. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 370–387. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_23
19. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.J.: Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: ICSE, pp. 191–200. IEEE Computer Society/ACM (1994)
20. Jones, J.A., Harrold, M.J.: Empirical evaluation of the Tarantula automatic fault-localization technique. In: ASE, pp. 273–282. ACM (2005)
21. Jose, M., Majumdar, R.: Bug-assist: assisting fault localization in ANSI-C programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 504–509. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_40
22. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI, pp. 437–446. ACM (2011)
23. Moon, S., Kim, Y., Kim, M., Yoo, S.: Ask the mutants: mutating faulty programs for fault localization. In: ICST, pp. 153–162. IEEE Computer Society (2014)
24. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. TOSEM **20**, 11:1–11:32 (2011)
25. Papadakis, M., Le Traon, Y.: Metallaxis-FL: mutation-based fault localization. Softw. Test. Verif. Reliab. **25**, 605–628 (2015)
26. Pearson, S., et al.: Evaluating and improving fault localization. In: ICSE, pp. 609–620. IEEE Computer Society/ACM (2017)
27. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39. IEEE Computer Society (2003)
28. Santelices, R.A., Jones, J.A., Yu, Y., Harrold, M.J.: Lightweight fault-localization using multiple coverage types. In: ICSE, pp. 56–66. IEEE Computer Society (2009)
29. Schäf, M., Schwartz-Narbonne, D., Wies, T.: Explaining inconsistent code. In: ESEC/FSE, pp. 521–531. ACM (2013)
30. Shen, S.Y., Qin, Y., Li, S.K.: Minimizing counterexample with unit core extraction and incremental SAT. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 298–312. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_20
31. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**, 121–189 (1995)
32. Wang, C., Yang, Z., Ivančić, F., Gupta, A.: Whodunit? Causal analysis for counterexamples. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 82–95. Springer, Heidelberg (2006). https://doi.org/10.1007/11901914_9
33. Weiser, M.: Program slicing. In: ICSE, pp. 439–449. IEEE Computer Society (1981)

34. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. Trans. Reliab. **63**, 290–308 (2014)
35. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. TSE **42**, 707–740 (2016)
36. Zeller, A.: Isolating cause-effect chains from computer programs. In: FSE, pp. 1–10. ACM (2002)