Fault Localization on Verification Witnesses

Dirk Beyer, Matthias Kettl, and Thomas Lemberger

LMU Munich, Germany

Abstract. When verifiers report an alarm, they export a violation witness (exchangeable counterexample) that helps validate the reachability of that alarm. Conventional wisdom says that this violation witness should be very precise: the ideal witness describes a single error path for the validator to check. But we claim that verifiers overshoot and produce large witnesses with information that makes validation unnecessarily difficult. To check our hypothesis, we reduce violation witnesses to that information that automated fault-localization approaches deem relevant for triggering the reported alarm in the program. We perform a large experimental evaluation on the witnesses produced in the International Competition on Software Verification (SV-COMP 2023). It shows that our reduction shrinks the witnesses considerably and enables the confirmation of verification results that were not confirmable before.

1 Introduction

Nowadays, a large body of automated formal-verification tools is available [4]. But when these tools present a verification alarm to the user without any additional information, it requires expensive reasoning about the program. So when a modern verifier reports an alarm, it also produces a *violation witness* [7] that describes a set of paths through the program of which at least one provokes the reported alarm. This allows to validate the claimed alarm with a separate, independent validator.[1] Developers may also use witnesses to debug. The description of paths through the program can convey more information than a single test input. For example, a violation witness may describe a full range of input values that trigger an alarm, or describe that the found program paths that trigger the alarm all pass through a specific code location.

A trivial violation witness that describes all program paths is valid, but it is not helpful. Conventional wisdom in the community assumes that a precise witness that only describes a single program path is the most helpful and the easiest to check for a validator. We observe that this makes some verifiers produce

A poster version of this paper is published in Proc. ICSE 2024 [15].

[1] All participants of the International Competition on Software Verification (SV-COMP) [4] produce violation witnesses. SV-COMP validates them and expects that this is significantly faster than the original verification: participating verifiers get 900 s of CPU time to analyze a program, but the verifier only receives points if a validator can confirm the produced violation witness within 90 s.

large witnesses that describe details about a program path that are not relevant with regards to the reported alarm. For example, an overly-detailed witness may describe a concise sequence of loop iterations in the program, even though the program reaches the alarm independent of that loop. Such details may slow down the validator when reasoning about the violation.

As a countermeasure, we propose to reduce witnesses to a selection of relevant information with fault localization. Given an error path, fault localization reports the statements that it suspects responsible for the reachability of that path. This makes witnesses more comprehensive for both machines and humans. We use three different fault-localization techniques based on SMT formulas: MaxSat [29], its derivative MinUnsat, and the baseline Unsat where we ask for an arbitrary set of statements that are relevant for the reachability of an error path.

We perform large-scale experiments on the data of SV-COMP 2023. These show that our reduction does not influence the quality of the witnesses, but reduces their size by 25 %. In addition, most validators confirm more alarms after our reduction.

Contributions. We make the following contributions:

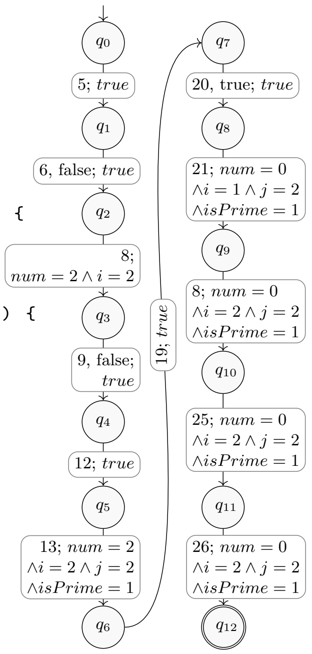
- We design the first formal approach to apply fault localization to software verification witnesses (violation witnesses) [8].
- Our implementation is available open source. Fault localization is implemented as part of the verification framework CPAchecker [10], and witness reduction is implemented in the new tool Flow [14].
- We perform a large experimental evaluation that shows that the approach can effectively reduce the size of the witnesses (by 25 %) and that the confirmation rate of validators is improved through this (by up to 36 %). We use 21 356 original witnesses that were produced by 14 verifiers on 3 225 verification tasks during SV-COMP 2023.
- Our code and all experimental data are available in an evaluated reproduction artifact [11].

Example. Figure 1 shows an example program that computes the prime factors for a number `num` that is provided through function `nondet()`. If a factor `i` is found for the number (`num % i == 0`) (line 9), it is checked whether `i` is a prime number (lines 12 to 18). If it is (`isPrime = 1`), the number `num` is divided by `i` and the process is repeated until `num` is 1 (lines 19 to 22). This way, all prime factors for `num` could be found. But the program has a fault in the computation of the division (line 20): Instead of computing `num = num / i`, there is an off-by-one error: `num = num / (i + 1)`. Therefore, the program calls `reach_error()` for input `num = 2`: The first iteration of the for-loop in line 8 assigns `num = 0` instead of the correct `num = 1`.

Assume we do not know that this error exists and we want to know whether a call to `reach_error()` is reachable. When we run the formal verifier UAutomizer [24, 25] on the program with property "`reach_error` is never called", it reports an alarm. It also provides a violation witness that represents at least one claimed counterexample to the property. This violation witness is presented as

```
1  int nondet();
2  void reach_error();
3
4  void main() {
5   int num = nondet();
6   if (num < 1) return;
7
8   for (int i=2; i <= num; i++) {
9    if (!(num % i == 0)) {
10    continue;
11   }
12   int isPrime = 1;
13   for (int j=2; j <= i/2; j++) {
14    if (i % j == 0) {
15     isPrime = 0;
16     break;
17    }
18   }
19   if (isPrime) {
20    num = num / (i + 1);
21    i--;
22   }
23  }
24
25  if (num != 1) {
26   reach_error();
27  }
28 }
```

Fig. 1: Program that computes factorization, with an off-by-one error in line 20 (underlined)



q_0
5; *true*
q_1
6, false; *true*
q_2
8; $num = 2 \land i = 2$
q_3
9, false; *true*
q_4
12; *true*
q_5
13; $num = 2 \land i = 2 \land j = 2 \land isPrime = 1$
q_6
19; *true*
q_7
20, true; *true*
q_8
21; $num = 0 \land i = 1 \land j = 2 \land isPrime = 1$
q_9
8; $num = 0 \land i = 2 \land j = 2 \land isPrime = 1$
q_{10}
25; $num = 0 \land i = 2 \land j = 2 \land isPrime = 1$
q_{11}
26; $num = 0 \land i = 2 \land j = 2 \land isPrime = 1$
q_{12}

Fig. 2: Detailed violation witness that describes the program path to the violation in line 26

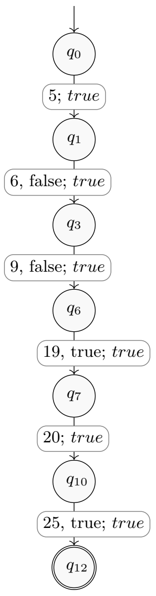


Fig. 3: Less detailed violation witness, reduced with our approach

automaton in Fig. 2. The witness describes a detailed sequence of source-code lines and variable values that it claims an execution must pass to reach a specification violation. Transition labels are denoted as pairs of a syntax guard and state-space guard, divided by a semicolon. Syntax guards name a line number and, optionally, after a comma, a branching condition true or false. State-space guards describe the required program state space at the specific transition.

All states ($q_0, \ldots, q_{12}$) have an implicit self-transition that is matched only if no other outgoing transition is valid for an execution step. To transition from q_2 to q_3, for example, the values of *num* and i have to be equal to 2. Otherwise, execution stays in q_2. The witness in Fig. 2 is very detailed and describes more information than necessary. This may limit a witness validator that tries to reason about the property violation, because the witness steers the validator to a precise path.

The strong contrast to this is a violation witness that enters an accepting state on every transition. It does not contain any information and does not restrict the program executions; it just says that there is a property violation *somewhere* in the program. It fits any violation that a validator finds.

We aim for the middle of these two extremes: We reduce the witness to only include information about the program that do not unnecessarily restrict the state-space, but restrict it to (a subset of) executions that actually reach the error.

To do this, we run a fault-localization approach on the counterexample described by the witness and produce a new, reduced witness (Fig. 3). This reduced witness only contains those transitions of the original witness that the fault localization deems relevant for the feasibility of the counterexample. In our example, this is the following lines: line 5, which initializes `num`, line 6, which restricts the value of `num`, line 9, which instructs analysis to continue with line 12, line 19, which instructs analysis to continue with line 20, line 20, which is the erroneous statement, and line 25, which guards the call to `reach_error()`. The reduced witness is significantly smaller and still contains relevant information for reasoning about the witness.

Related Work. Execution-based witness validation [9] eases the use of a violation witness by turning it into an executable test. A developer can then debug this test in the used manner to reason about the reported violation. Other approaches also exist for the execution of counterexamples [6, 22], but these do not use a common exchange format. Other approaches [20, 33, 36] provide simulators for counterexamples that allow to directly "execute" a counterexample step-by-step, similar to prominent debuggers for program execution. Test execution and execution-simulation are orthogonal to our approach: with an executable test, the developer has the full program path to debug, and is not hinted towards statements of potentially high interest. Additionally, error paths that are not described by the test case are lost.

Multiple fault-localization approaches exist, based on code-coverage statistics [1, 27, 38], slicing-based approaches [26], and logic reasoning [19, 23, 28, 29].

There exists a preliminary study [3] on the structure of verification witnesses, but it focuses on witnesses for correctness proofs and does not examine the indication of violation-witness detail on their confirmation rate. We close this gap with our new approach.

Performing fault localization on witnesses has been studied before [30] in the context of timed automata. Given a counterexample, the approach computes the maximum satisfiable core of the trace and tries to fix the automaton.

2 Background

Program Representation. For the sake of presentation, we consider an imperative, sequential programming language with two types of operations: assign operation (`x = x + 1`) and assume operation (`[x <= 0]`). We use assign operation `x = nondet()` to signal introduction of a new non-deterministic value, and special statement `reach_error()` to signal an error.[2] All program variables are of type integer.[3] Set X is the set of all program variables in a program and set Ops is the set of all possible program operations over X.

[2] All safety properties on programs can be reduced to this property.

[3] Our implementation supports the GNU C programming language.

We represent programs as control-flow automata (CFAs). A CFA $P = (L, l_0, G)$ consists of the set of program locations L, the initial program location $l_0 \in L$, and a set of control-flow edges $G \subseteq L \times \mathit{Ops} \times L$.

We use formulas in first-order logic. Given a symbol s, we define $s^{\langle i \rangle}$ as the instantiation of s with i primes. For example, $s^{\langle 0 \rangle} = s$ and $s^{\langle 2 \rangle} = s''$.

The set $\mathcal{C}$ contains all possible concrete program states. A concrete program state $c : X \mapsto \mathbb{Z}$ consists of one value assignment for each program variable. A program path $pp = l_0 \xrightarrow{op_0} \dots \xrightarrow{op_n} l_{n+1}$ is a run through the CFA so that $(l_i, op_i, l_{i+1}) \in G$. A program execution $ex(pp, c_0) = (l_0, c_0) \xrightarrow{op_0} \dots \xrightarrow{op_n} (l_{n+1}, c_{n+1})$ for program path pp and initial state $c_0 \in \mathcal{C}$ is feasible iff each c_{i+1} is a valid product of evaluating op_i on c_i. Trace formula $\mathrm{TF}(pp)$ is a sequence of formulas that exactly describes the valid program states in all possible program executions for pp. For example, $pp = l_5 \xrightarrow{\texttt{num = nondet()}} l_6 \xrightarrow{\texttt{[!(num < 1)]}} l_{8_0} \xrightarrow{\texttt{i = 2}} l_{8_1} \xrightarrow{\texttt{[i <= num]}} \dots \xrightarrow{\texttt{i = i + 1}} l_{8_1} \xrightarrow{\texttt{[!(i <= num)]}} l_{25} \xrightarrow{\texttt{[num != 1]}} l_{26}$ is represented by $\mathrm{TF}(pp) = \langle num^{\langle 0 \rangle} = nondet^{\langle 0 \rangle}, \neg(num^{\langle 0 \rangle} < 1), i^{\langle 0 \rangle} = 2, i^{\langle 0 \rangle} \leq num^{\langle 0 \rangle}, \dots, i^{\langle 2 \rangle} = i^{\langle 1 \rangle} + 1, \neg(i^{\langle 2 \rangle} \leq num^{\langle 1 \rangle}), num^{\langle 1 \rangle} \neq 1 \rangle$. We define $\mathrm{TF}(pp)\{i\}$ as the i-th formula in $\mathrm{TF}(pp)$, starting with index 0. We expect that, for each operation op_i in pp, there is exactly one corresponding formula $\mathrm{TF}(pp)\{i\}$. We define subset $\mathrm{TF}(pp)\{:j\} = \big\{\mathrm{TF}(pp)\{0\}, \dots, \mathrm{TF}(pp)\{j\}\big\}$ as the prefix of $\mathrm{TF}(pp)$ up to index j.

A counterexample is a finite program path $l_0 \xrightarrow{op_0} \dots \xrightarrow{op_{n-1}} l_n \xrightarrow{\texttt{reach\_error()}} l_{n+1}$ that ends with an operation `reach_error()`. We say that a counterexample cex is feasible if there exists a $c_0 \in \mathcal{C}$ so that $ex(cex, c_0)$ is feasible. For the sake of our presentation, we assume that a `reach_error()` is always directly preceded by a single assume operation op_{n-1}.[4]

We represent program properties as *observer automata*: A program property φ for a program $P = (L, l_0, G)$ is a finite-state automaton $\varphi = (\Omega, \Sigma_\varphi, \delta, \omega_0, F)$ with states Ω, alphabet $\Sigma_\varphi = 2^G$, transitions $\delta \subseteq \Omega \times \Sigma_\varphi \times \Omega$, initial state $\omega_0 \in \Omega$ and accepting states $F \subseteq \Omega$, where $\omega_0 \notin F$. The accepting states F represent a violation to the property. The observer automaton gets as input a program path.

Fault Localization. A *suspect* is a subset $f \subseteq G$ of CFA edges whose operations may be responsible for a feasible counterexample.

Given a feasible counterexample cex, fault localization has the goal to determine a set $\mathcal{F} = \{f_0, f_1, \dots\}$ of suspects.

Because cex is feasible, we know that the conjunction $\bigwedge \mathrm{TF}(cex)$ of the trace formula's elements is satisfiable. We define the *precondition* ψ, the *faulty trace* π, and the *post condition* ϕ. The precondition $\psi = (nondet^{\langle 0 \rangle} = v_0 \wedge \dots \wedge nondet^{\langle k \rangle} = v_k)$ describes one satisfying variable assignment of $\bigwedge \mathrm{TF}(cex)$ for all non-deterministic values (variables $nondet^{\langle i \rangle}$) that occur in $\bigwedge \mathrm{TF}(cex)$. It can be generated by extracting the relevant variable assignments from a model of $\bigwedge \mathrm{TF}(cex)$. For example, one precondition for the counterexample described by Fig. 2 is $\psi = (nondet^{\langle 0 \rangle} = 2)$. It initializes $num = 2$.

[4] In our implementation, we find the last assume operation before the error location and of these we use all assume operations that originate from the same code line.

Algorithm 1 MaxSat(ψ, π, ϕ)

```
Input: Precondition $\psi$, faulty trace $\pi$, postcondition $\phi$
Output: Set $\mathcal{F}$ of candidate faults
1: $\mathcal{F} = \{\}$
2: $susp = 2^{\pi}$
3: for $size$ in $1 \ldots |\pi|$ do
4:     $\mathcal{F} = \mathcal{F} \cup \{f \in susp \mid size = |f| \text{ and } \psi \wedge \bigwedge(\pi \setminus f) \wedge \phi \text{ sat}\}$
5:     $susp = \{t \in susp \mid \forall f \in \mathcal{F}.\ f \not\subseteq t\}$
6: return $\mathcal{F}$
```

Algorithm 2 MinUnsat(ψ, π, ϕ)

```
Input: Precondition $\psi$, faulty trace $\pi$, postcondition $\phi$
Output: Set $\mathcal{F}$ of candidate faults
1: $susp = 2^{\pi}$
2: $\mathcal{F} = \{\}$
3: for $size$ in $1 \ldots |\pi|$ do
4:     $\mathcal{F} = \mathcal{F} \cup \{f \in susp \mid size = |f| \text{ and } \psi \wedge \bigwedge f \wedge \phi \text{ unsat}\}$
5:     $susp = \{t \in susp \mid \forall f \in \mathcal{F}.\ f \not\subseteq t\}$
6: return $\mathcal{F}$
```

The faulty trace $\pi = \text{TF}(cex)\{: n-2\}$ is the prefix of $\text{TF}(cex)$ that includes all possible program faults. It excludes the formula of the last assume operation op_{n-1}. The post condition $\phi = \neg\text{TF}(cex)\{n-1\}$ represents the final assume operation that guards the `reach_error()`. The `reach_error()` is reachable when $\neg\phi = \text{TF}(cex)\{n-1\}$ is fulfilled. For example, the postcondition for the counterexample that is described by Fig. 2 is $\phi = \neg(num \neq 1)$. If $\neg\phi = num \neq 1$ is fulfilled, the error location is reached. When formula $\psi \wedge \bigwedge \pi \wedge \neg\phi$ represents a feasible counterexample, we can assume that it is satisfiable. Because ψ defines a definite initial assignment for all variables in ψ and π, we can then also assume that the formula $\psi \wedge \bigwedge \pi \wedge \phi$ with a fulfilled postcondition ϕ is unsatisfiable.

MaxSAT. MaxSat [29] (Alg. 1) computes a set of candidate faults for a counterexample by finding all minimal combinations of program operations that together contribute to the feasibility of the counterexample.

MaxSat initializes the set *susp* with all subsets of π. This are all suspects for a program fault. Then, MaxSat starts with the smallest possible candidate subsets that only consist of a single clause ($size = 1$) and increasingly considers candidate subsets of larger *size*. For each *size*, MaxSat selects all suspects $f \subseteq \pi$ that consist of *size* clauses and for which $\psi \wedge \bigwedge(\pi \setminus f) \wedge \phi$ is satisfiable. The set *susp* of possible candidate subsets is then updated to only include subsets of π that are not supersets of any previously selected subset. After all relevant combinations of program operations have been considered, the set $\mathcal{F}$ of selected subsets is returned as the set of possible suspects.

MinUnsat. Analogous to MaxSat, algorithm MinUnsat (Alg. 2) computes a set of candidate faults for a counterexample by computing all subsets $f \subseteq \pi$ that suffice so that $\psi \wedge \bigwedge f \wedge \phi$ is unsatisfiable, and for which no smaller subset $f' \subset f$ fulfills the same criterion.

Unsat. Last, we use algorithm Unsat as a baseline for fault localization: Unsat asks an SMT solver for an arbitrary subset $f \subseteq \pi$ that makes $\psi \wedge \bigwedge f \wedge \phi$ unsatisfiable. This may even be π itself.

Example. To illustrate MaxSat and MinUnsat, assume the precondition $\psi = (nondet^{\langle 0\rangle} = 2)$, the trace π with

$$\begin{aligned}\pi = \{&num^{\langle 0\rangle} = nondet^{\langle 0\rangle},\ \neg(num^{\langle 0\rangle} < 1),\ i^{\langle 0\rangle} = 2,\ i^{\langle 0\rangle} \le num^{\langle 0\rangle},\\ &num^{\langle 0\rangle}\% i^{\langle 0\rangle} = 0,\ isPrime^{\langle 0\rangle} = 1,\ j^{\langle 0\rangle} = 2,\ \neg(j^{\langle 0\rangle} \le num^{\langle 0\rangle}/2),\\ &isPrime^{\langle 0\rangle} \ne 0,\ num^{\langle 1\rangle} = num^{\langle 0\rangle}/(i^{\langle 0\rangle}+1),\ i^{\langle 1\rangle} = i^{\langle 0\rangle} - 1,\\ &i^{\langle 2\rangle} = i^{\langle 1\rangle} + 1,\ \neg(i^{\langle 2\rangle} \le num^{\langle 1\rangle})\}\end{aligned}$$

and the postcondition $\phi = (num^{\langle 1\rangle} = 1)$. Given these three, MaxSat produces the set $\mathcal{F} = \{\{num^{\langle 1\rangle} = num^{\langle 0\rangle}/(i^{\langle 0\rangle}+1)\}, \{num^{\langle 0\rangle} = nondet^{\langle 0\rangle}\}, \{i^{\langle 0\rangle} = 2\}\}$ of three suspects, because it is enough to remove any of the three assignments f from the trace to make the formula $\psi \wedge \bigwedge(\pi \setminus f) \wedge \phi$ satisfiable. MinUnsat produces the set $\mathcal{F} = \{\{num^{\langle 1\rangle} = num^{\langle 0\rangle}/(i^{\langle 0\rangle}+1), num^{\langle 0\rangle} = nondet^{\langle 0\rangle}, i^{\langle 0\rangle} = 2\}\}$ of a single suspect f, because all three assignments are necessary to make the formula $\psi \wedge \bigwedge f \wedge \phi$ unsatisfiable.

To make fault localization terminate faster, we introduce an option to immediately return the first found suspect, instead of collecting all.

Violation Witness. Given a CFA (L, l_0, G), a *source-code guard* $S \subseteq G$ is a set of control-flow edges. A *state-space guard* p is a Boolean assumption over program variables X. The type of state-space guards is Φ. A violation-witness automaton [8] is a non-deterministic finite-state automaton $W = (Q, \Sigma_W, \delta, q_0, F)$ with the following components: States Q, alphabet $\Sigma_W = 2^G \times \Phi$, transitions $\delta \subseteq Q \times 2^G \times \Phi \times Q$, initial state $q_0 \in Q$, and accepting states $F \subseteq Q$. The violation-witness automaton gets as input a program execution. For program-execution step $(l, c) \xrightarrow{op_i} (l', c')$, a transition $(q_i, (S, p), q_j)$ is possible if $(l, op_i, l') \in S$ and if c' is a satisfying assignment for p; i.e., formula $p \wedge \bigwedge_{x \in X} x = c'(x)$ is satisfiable. Each state also includes an implicit 'otherwise' self-transition, which only activates if there is no other matching transition from the current state. A violation-witness automaton describes a set of counterexamples by restricting the set of all possible program executions through source-code guards and state-space guards: Source-code guards restrict the control flow, and state-space guards restrict the potential program states. In SV-COMP, a violation witness[5] is described in the `GraphML` format [16], which is based on XML.

Witness Validation. Given a program P, a specification φ, and a violation witness W, a witness validator [7] checks whether W accepts any program execution on P that violates φ. If it does, W is *confirmed*. Otherwise it is *rejected*.

[5] https://gitlab.com/sosy-lab/benchmarking/sv-witnesses/-/blob/0ca5dbf/doc/README-GraphML.md

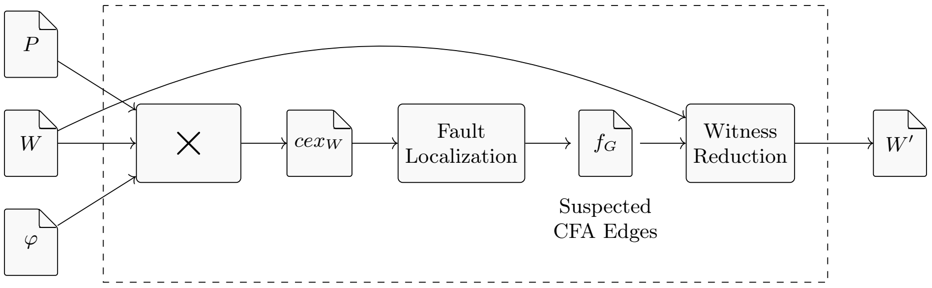


Fig. 4: Workflow of fault localization on violation witnesses. By creating the product automaton of program P, witness W, and specification φ, we obtain counterexample cex_W and apply fault localization to find suspects f_G. With these, witness W is reduced to W'.

3 Fault Localization on Violation Witnesses

Figure 4 illustrates our approach. Given a CFA $P = (L, l_0, G)$, a violation-witness automaton $W = (Q, \Sigma_W, \delta_W, q_0, F_W)$, and a program property $\varphi = (\Omega, \Sigma_\varphi, \delta_\varphi, \omega_0, F_\varphi)$, we (1) reconstruct a counterexample cex_W from W, (2) compute suspects f_G through fault localization on cex_W, and (3) produce a new violation-witness automaton W' that only contains those states and transitions of W that match the program operations contained in f_G.

Reconstruct Counterexample. To reconstruct a counterexample from the witness, we first build the product automaton $P \times W \times \varphi = (Q_\times, \Sigma_\times, \delta_\times, q_{\times 0}, F_\times)$, with states $Q_\times = L \times Q \times \Omega$, alphabet $\Sigma_\times = Ops \times \Sigma_W \times \Sigma_\varphi$, transition relation $\delta_\times$, and initial state $q_{\times 0} = (l_0, q_0, \omega_0)$. The transition relation $\delta_\times$ contains transition $\big(q_\times, (op, (S_W, p), S_\varphi), q'_\times\big)$ with $q_\times = (l, q, \omega)$ and $q'_\times = (l', q', \omega')$ if the following three conditions hold: $(l, op, l') \in G, (q, (S_W, p), q') \in \delta_W, (\omega, S_\varphi, \omega') \in \delta_\varphi$. The product automaton accepts all states for which the violation-witness automaton W and observer automaton φ agree on a violation. Formally, the set of accepting states is $F_\times = \{(l, q, \omega) \mid q \in F_W, \omega \in F_\varphi\}$.

The product automaton gets as input a program execution. For the program-execution step $(l, c) \xrightarrow{op_i} (l', c')$, a transition $\big(q_\times, (op, (S_W, p), S_\varphi), q'_\times\big) \in \delta_\times$ from state $q_\times = (\mathtt{l}, q, \omega)$ to state $q'_\times = (\mathtt{l}', q', \omega')$ is possible if all of the following conditions hold: $op_i = op, l = \mathtt{l}, l' = \mathtt{l}', (l, op_i, l') \in S_W \cap S_\varphi,$ and c' fulfills p.

The product automaton's accepting runs describe all counterexamples that are described by the violation-witness automaton, that exist in the program under analysis, and that violate the specification. From these, we select one arbitrary run and use the program-path information of that run as counterexample cex_W.

Fault Localization. We apply fault localization to counterexample cex_W. This returns the set $\mathcal{F}$ of suspects. From that set, we consider the suspect $f = \{\mathrm{TF}(cex_W)\{i\}, \ldots, \mathrm{TF}(cex_W)\{k\}\}$ with the smallest size and map it to the set $f_G = \{(l_i, op_i, l'_i), \ldots, (l_k, op_k, l'_k)\}$ of corresponding, suspected CFA edges. If

Algorithm 3 reduce$_c(W, f_G)$

```
Input: Violation-witness automaton $W = (Q, \Sigma, \delta, q_0, F)$,
    relevant CFA edges $f_G = \{(l_i, op_i, l'_i), \dots, (l_k, op_k, l'_k)\}$
Output: Reduced violation-witness automaton $W'$
 1: $\delta_S = \{(q, (S, p), q') \in \delta \mid S \cap f_G \neq \emptyset\}$
 2: if $c = full$ then
 3:     $\delta_{nop} = \{(q, (G, true), q') \mid (q, (S, p), q') \in \delta \setminus \delta_S\}$
 4: if $c = \Phi$ then
 5:     $\delta_{nop} = \{(q, (S, true), q') \mid (q, (S, p), q') \in \delta \setminus \delta_S\}$
 6: $\delta_{|f_G} = \delta_S \cup \delta_{nop}$
 7: $W' = (Q, \Sigma, \delta_{|f_G}, q_0, F)$
 8: return $W'$
```

Algorithm 4 collapse(W)

```
Input: Violation-witness automaton $W = (Q, \Sigma, \delta, q_0, F)$
Output: Collapsed violation-witness automaton $W_c$
 1: $Q_c = Q$
 2: $\delta_c = \delta$
 3: $F_c = F$
 4: $waitlist = \{q' \in Q \mid (q_0, (\cdot, \cdot), q') \in \delta\}$
 5: while $waitlist \neq \emptyset$ do
 6:     choose $q' \in waitlist$
 7:     $waitlist = waitlist \setminus \{q'\}$
 8:     if $\{(q, (S, p), q') \in \delta \mid S \neq G \vee p \neq true\} \neq \emptyset$ then
 9:         continue
10:     $\delta_{q'|nop} = \{(q, (G, true), q') \in \delta_{nop}\}$
11:     if $|\delta_{q'|nop}| > 1 \vee q' = q_0$ then
12:         continue
13:     $Q_c = Q_c \setminus \{q'\}$
14:     $\delta_{del} = \{(q', (S, p), q'') \in \delta_c\}$
15:     $\delta_{adj} = \{(q, (S, p), q'') \mid (q', (S, p), q'') \in \delta_{del}\}$
16:     $\delta_c = \delta_c \cup \delta_{adj} \setminus (\delta_{q'|nop} \cup \delta_{del})$
17:     if $q' \in F$ then
18:         $F_c = F_c \cup \{q\} \setminus \{q'\}$
19:     $waitlist = waitlist \cup \{q'' \mid (q, (\cdot, \cdot), q'') \in \delta_c\}$
20: $W_c = (Q_c, \Sigma, \delta_c, q_0, F_c)$
21: return $W_c$
```

multiple suspects of the same size exist, an arbitrary suspect is selected. The suspected CFA edges f_G are used to reduce the violation witness.

Witness Reduction. We define three variants of witness reduction. Reduction $\mathsf{r}_{\mathsf{state}}(W, f_G) = \text{reduce}_{\Phi}(W, f_G)$ deletes all state-space guards that are deemed irrelevant by fault localization, but keeps all source-code guards. Reduction $\mathsf{r}_{\mathsf{match}}(W, f_G) = \text{reduce}_{full}(W, f_G)$ deletes all state-space guards and source-code guards that are deemed irrelevant by fault localization. Finally, reduction

$\mathsf{r_{all}}(W, f_G) = \text{collapse}(\text{reduce}_{full}(W, f_G))$ reduces the witness like $\mathsf{r_{match}}$, but also collapses the resulting witness to produce a smaller violation witness.

Algorithm 3 describes methods reduce_{Φ} and reduce_{full}: The algorithm first selects those transitions $\delta_S \subseteq \delta$ for which at least one edge in f_G matches the transition's source-code guard S. It then selects all remaining transitions $\delta \setminus \delta_S$. Reduction reduce_{full} then replaces the source-code guards with G (we forget the source-code guard and match everything) and state-space guards with $true$ (line 3). Reduction reduce_{Φ} (line 5) only replaces the state-space guards with $true$, but keeps the source-code guards to steer the witness validation. In both cases, we call the resulting set δ_{nop}.

After the initial reduction, reduction $\mathsf{r_{all}}$ collapses the witness (Alg. 4). The intuition behind this is that transitions with the trivial guards $(G, true)$ still restrict the set of program paths because they define a minimum number of program operations that must occur between two transitions with non-trivial guards. We remove this implicit restriction on a best-effort basis. We do not perform a precise elimination of δ_{nop} because this may greatly alter the structure of the automaton. Instead, we only collapse sequential sequences of transitions: Algorithm 4 first initializes the states Q_c after collapse, the transitions δ_c after collapse, and the accepting states F_c after collapse, with the original values of W. It then starts traversal of the violation-witness automaton W at all direct successors of q_0. For each visited automaton state q' (lines 6-7), Alg. 4 checks whether q' has any ingoing transition with a non-trivial guard (line 9), has more than one ingoing transition, or is the entry state (line 11). In both cases, Alg. 4 continues with the next state. Otherwise, q' only has a single ingoing edge with only trivial guards, and it can be collapsed into its successors q''. First, q' is removed from Q_c (line 13). Next, Alg. 4 deletes the ingoing and outgoing transitions ($\delta_{q'|nop}$ and δ_{del}) of q' from δ_c, and instead adds new transitions that directly go from q to q'' (lines 14–16). If q' is removed from Q_c and it is an accepting state (line 17), then its predecessor q becomes an accepting state instead (line 18). Algorithm 4 then continues with all successor states (line 19). When no more states can be removed, Alg. 4 exhausts the waitlist by running into line 9 or line 12 for all states. Finally, the violation-witness automaton W_c is defined and returned.

To ensure that the reduced violation witness describes the same set of program paths as the original witness, we do not remove source-code guards that restrict the control flow. In particular, we always keep source-code guards in the `GraphML` witness that contain one of the following keys: `control`, `enterLoopHead`, `enterFunction`, or `returnFromFunction`. Our replication package [11] provides experimental results for a witness reduction that removes all source-code guards. The data shows that this reduction is not beneficial since the produced witnesses have a low confirmation rate.

Soundness. Algorithm 3 produces a sound overapproximation W' of W. Both the removal of source-code guards and state-space guards can only increase the set of program paths that are described by a violation-witness automaton, due to the implicit otherwise-self loops at each automaton state. In consequence, the reduced witness W' is an overapproximation: it describes a superset of the

Table 1: Data on the reduction with $\mathsf{r}_{\mathsf{all}}$ and the fault-localization techniques

		Number of witnesses where fault localization was successful				Average number of edges with		
Producer	Total	MaxSat	MinUnsat	Unsat	Union	MaxSat	MinUnsat	Unsat
2LS	490	134	112	93	138	409	256	728
Bubaak	1 464	1 048	870	267	1 124	17	16	102
CBMC	2 256	605	505	105	630	438	220	466
CPAchecker	1 932	1 276	1 201	340	1 386	762	697	3 269
ESBMC-kind	2 164	587	565	243	716	36	26	4 810
Graves	2 065	1 266	1 097	360	1 405	974	612	4 365
PeSCo	2 147	855	683	269	944	809	670	2 837
Symbiotic	1 418	1 076	897	190	1 088	15	15	25
UAutomizer	1 060	380	343	205	390	272	167	641
UKojak	379	184	179	96	184	85	72	47
UTaipan	869	331	302	177	336	293	189	619
VeriAbs	1 588	404	363	128	438	701	595	2 321
VeriAbsL	1 777	447	410	134	485	686	625	2 300
VeriFuzz	1 747	801	673	144	863	18	24	42
Overall	21 356	9 394	8 200	2 801	10 127	4 379	3 232	18 513
Mean	1 525	671	586	200	723	394	299	1 612
Median	1 668	596	535	184	673	351	205	685

program paths that are described by the original witness W. If the original witness describes a feasible counterexample, then the reduced witness describes the same counterexample (and maybe more). Analogous, Alg. 4 produces a sound overapproximation because the removal of violation-witness-automaton edges can only increase the set of program paths that are described by the violation-witness automaton.

4 Evaluation

We answer the following research questions:

RQ 1 Witness Reduction. Can fault localization reduce the number of transitions in violation-witness automata?

RQ 2 Confirmation Rate. Does the level of detail influence the confirmation rate of violation witnesses?

RQ 3 Reduction Variants. How do the different reduction variants $\mathsf{r}_{\mathsf{match}}$, $\mathsf{r}_{\mathsf{state}}$, and $\mathsf{r}_{\mathsf{all}}$ influence the confirmation rate?

4.1 Experiment Setup

We run all experiments on machines with an Intel Xeon E3-1230 v5 @ 3.40 GHz 8-core processor and 33 GB RAM. We limit the machines to only use 2 cores and 7 GB of RAM. We execute all our experiments with BenchExec v3.11[6], a reliable

[6] https://github.com/sosy-lab/benchexec/tree/637de81c0d

tool for benchmarking [12]. The above setup is equal to the setup of SV-COMP 2023 [4] to ensure comparability between the validation results. The timeout for witness validation is set to 90 s (SV-COMP standard).

We use all 3 225 non-recursive, unsafe verification tasks violating the `unreach-call` property (`reach_error` is reachable) in the sv-benchmarks[7] version of SV-COMP 2023. We use 14 non-hors concours participants in the *Reach-Safety* category. We exclude the verifiers Mopsa [32] and Goblint [37] since they did not produce any violation witnesses. We also exclude the verifier Theta [39] because Theta produces violation-witness automata with invalid source-code guards. We use all 21 356 violation witnesses [5] that the considered participants produce on the verification tasks.

We use the three fault-localization approaches MaxSat, MinUnsat, and Unsat. All three are implemented in CPAchecker and run with a time limit of 900 s. We use CPAchecker in revision 44191[8]. In CPAchecker, we use the SMT solver MathSAT5 [18]. CPAchecker exports a JSON format that describes all located suspects. We match these entries against the source-code guards of the witnesses. We implement the witness reduction algorithms in our tool Flow[9]. All tools, scripts, and evaluation data are available in our replication package [11]. To validate our witnesses, we use the SV-COMP 2023 versions of CPAchecker [10], UAutomizer [25], MetaVal [13], and Symbiotic-Witch [2].

The following evaluation only considers those witnesses for which at least one fault-localization approach works. The left half of Table 1 shows the success rate of the three fault-localization approaches (columns) on each producer's witnesses (rows). Looking at the mean and median values on the bottom of the table, we find that Unsat only reduces approximately 15 % (2 801 of 21 356) of the witnesses. Asking MathSAT5 for an arbitrary UNSAT core fails more often than repeatedly checking for satisfiability of smaller subsets. However, we will later see that for large programs Unsat works better as checking many subsets becomes expensive. MaxSat and MinUnsat, on average, successfully reduce a third of the witnesses. Fault localization is not guaranteed to succeed in some cases, e.g., if we have to deal with pointers, or if the SMT solver times out when querying for (un)satisfiability. Column 'union' displays the number of witnesses where at least one fault localization technique was successful. In total, we successfully apply fault localization to 10 127 different witnesses and consider $3 \cdot (9\,394 + 8\,200 + 2\,801) = 61\,185$ witnesses that were produced across the three reduction approaches.

4.2 Experiment Results

RQ 1: Witness Reduction. Our approach significantly reduces the number of transitions of the original violation-witness automata. To show this, we use reduction variant $\mathsf{r}_{\mathsf{all}}$. We also observe a similar picture for variant $\mathsf{r}_{\mathsf{match}}$.

[7] `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp22`
[8] `https://svn.sosy-lab.org/software/cpachecker/trunk/?p=44191`
[9] `https://gitlab.com/sosy-lab/software/fault-localization-on-witnesses`

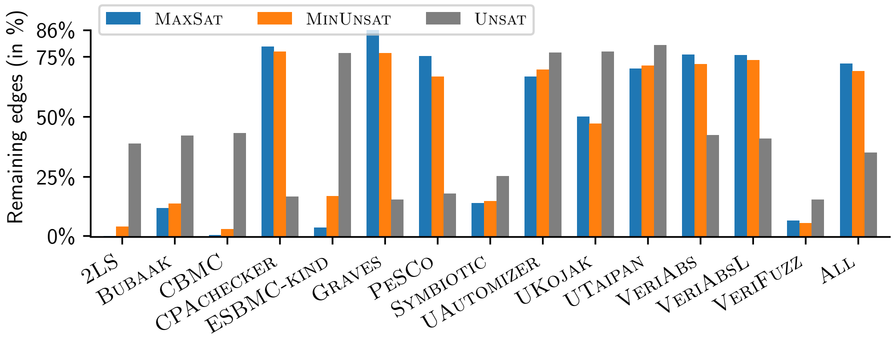


Fig. 5: Number of source-code guards in violation witnesses after applying reduction $\mathsf{r}_{\mathsf{all}}$ with the respective fault-localization approach; the plot shows the producers of the original witnesses on the x-axis; the lower the bars, the greater the reduction

Figure 5 shows the averaged reduction of transitions per verifier on successfully reduced tasks per producer. The x-axis names the tools from which we take the original witnesses. For every of the 14 producers we plot 3 bars corresponding to the three fault-localization approaches MaxSat, MinUnsat, and Unsat. A value of 25 % means that the original witnesses are 4 times as big as the reduced witnesses. The bars for one producer do not necessarily talk about the same witnesses because one technique may fail for a task where the others succeed. The lower the bar, the higher the reduction. The last column All indicates that Unsat has the highest reduction rate, followed by MinUnsat, and finally MaxSat. All shows the ratio of the sum of remaining edges after the reduction and the sum of edges in the original witnesses over *all* producers. Despite Unsat often keeping more edges for individual producers (e.g., 2ls and Bubaak), it significantly reduces larger witnesses (e.g., CPAchecker and PeSCo) causing less edges to survive on average. For the comparison, we exclusively consider witnesses where fault localization succeeded. Overall, fault localization has a significant impact on the reduction of violation witnesses. On average, MinUnsat and MaxSat remove approximately 30 % of the transitions for the vast majority of witnesses. Unsat reduces the witnesses even to one third of the original size. The right half of Table 1 shows the average number of edges per fault-localization approach (columns) for each verifier (rows). Columns MaxSat, MinUnsat, and Unsat show the average number of transitions in witnesses where the respective fault localization approach worked. We also see why Unsat has, in general, less remaining edges in the witnesses: the average number of transitions of witnesses where the approach works is higher. One reason is that it does not need to enumerate all possible subsets and can just return an arbitrary unsat core. Depending on which operations are marked relevant by fault localization reduced witnesses have different sizes although the unsat core marks equally many operations as relevant. Consider the program in

```
1  int x = 0;
2  while (x < 20) {
3    x = x + 1;
4  }
5  if (x == 20) {
6    assert(x % 5 != 0);
7  }
```

Fig. 6: Fault localization creates two minimal unsat cores: (1) the assignment `x = x + 1` and (2) the condition `x == 20` of the then-branch in line 5

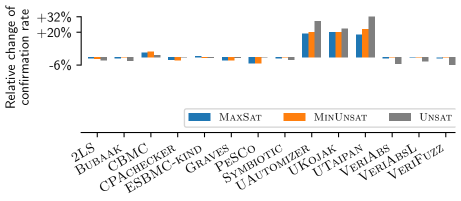
Figure 6. We initialize variable `x` with 0 and enter a while loop where we increment `x` 20 times. Eventually, we exit the loop with `x = 20` causing the program to reach an error location since 20 is divisible by 5. A detailed witness describes the complete path with 20 guards at line 3, the assumptions `x >= 20` and `x < 20` at line 2, and the assumption `x == 20` at line 5. Fault localization gives us 2 unsat cores: (1) `x = x + 1` at line 3, and (2) `x == 20` at line 5. Although all unsat cores are of same size, the reduction of the detailed witness varies. Unsat core (1) leaves 20 transitions while unsat core (2) removes all but one transition on line 5. This property of our reduction techniques causes the varying reduction percentage across different fault-localization approaches since a smaller set of suspects does not automatically lead to a higher reduction.

Yes, fault localization can significantly reduce the size of violation witnesses.

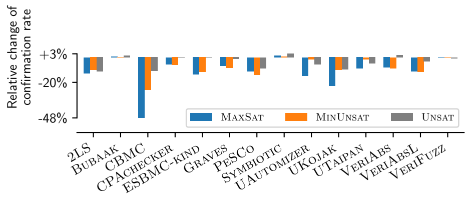
RQ 2: Confirmation Rate. Our approaches to witness reduction can have both positive and negative effects on the success of witness validation. For fairness reasons, we only evaluate the confirmation rate on witnesses that were deemed syntactically correct by the respective validator before and after reduction. To examine the effect, we consider, for each fault-localization approach, all original violation witnesses from 14 participants of SV-COMP 2023 for which the respective fault localization was successful. We use reduction variant $\mathsf{r}_{\mathsf{all}}$. From our data, we showcase the validators UAutomizer and MetaVal because they show the strongest positive and negative effect, respectively: MetaVal benefits the most from the reduction while UAutomizer performs significantly worse after reduction[10].

Figure 7a and Fig. 7b show the relative change of the confirmation rate after witness reduction with $\mathsf{r}_{\mathsf{all}}$ for the validators MetaVal and UAutomizer. Bars that reach positive values indicate a relative increase in the confirmation rate compared to the confirmation rate on the original witnesses. A value of 20 % means that after reduction, 20 % more of the considered witnesses can be confirmed by the respective validator. A value of −20 % means that after reduction, −20 % less of the considered witnesses can be confirmed by the respective validator.

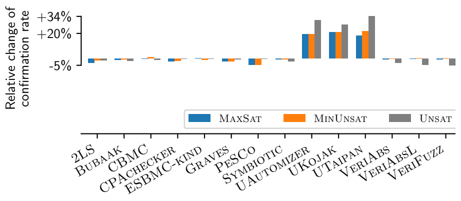
[10] The analysis of the other validators and all reduction approaches is available in our replication artifact [11].



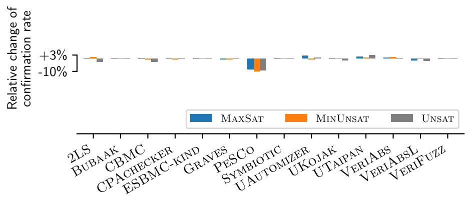
(a) MetaVal with $\mathsf{r}_{\mathsf{all}}$



(b) UAutomizer with $\mathsf{r}_{\mathsf{all}}$



(c) MetaVal with $\mathsf{r}_{\mathsf{match}}$



(d) UAutomizer with $\mathsf{r}_{\mathsf{state}}$

Fig. 7: Relative change of the confirmation rate of MetaVal and UAutomizer per fault-localization approach with different reduction variants

Figure 7a shows that our witness reduction has small negative effects (worst $-6\,\%$) on the confirmation rate of MetaVal for some witness producers, but also tremendous positive effects (best $32\,\%$) on the confirmation rate of MetaVal for some other witness producers. MetaVal uses verification engines as backend and annotates information in witnesses directly in the program. Therefore, giving the verifiers more freedom in analysis through more abstract violation witnesses benefits the MetaVal approach. In contrast, UAutomizer performs significantly worse for many fault-localization approaches when reduction variant $\mathsf{r}_{\mathsf{all}}$ is used. The confirmation rate decreases up to $-48\,\%$ after reduction. This decreases the strongest for fault-localization with MaxSat.

The confirmation rate of validators can also decrease, for two reasons: First, the reduction might remove all information and transform the new witness to the trivial witness. With trivial witnesses, validators have no other choice than performing a complete re-verification of the original program. We observe this for producers Bubaak [17], VeriFuzz [31], and Esbmc-kind [21] (cf. Table 1). Second, fault localization has no knowledge about the structure of the program and might therefore remove information that would exclude certain branches from being explored. This becomes especially visible when looking at the different reduction techniques. Technique $\mathsf{r}_{\mathsf{all}}$ has less negative effects on UAutomizer.

Yes. Depending on the reduction strategy, the confirmation rate of witness validators increases significantly on reduced witnesses.

RQ 3: Reduction Variants. In RQ 1 and RQ 2 we mainly focused on reduction variant $\mathsf{r}_{\mathsf{all}}$. In the following, we pick two examples to illustrate the effect that the reduction variant can have on the confirmation rate of reduced witnesses.

Figure 7c shows the confirmation rate of MetaVal with $\mathsf{r}_{\mathsf{match}}$ across the three considered fault-localization approaches and across all witness producers. It shows that MetaVal performs slightly better with $\mathsf{r}_{\mathsf{match}}$ compared to $\mathsf{r}_{\mathsf{all}}$ (Fig. 7a). The highest increase in confirmation rate is 34 %, compared to 32 % with $\mathsf{r}_{\mathsf{all}}$.

Figure 7d shows the confirmation rate of UAutomizer with $\mathsf{r}_{\mathsf{state}}$ across the three considered fault-localization approaches and across all witness producers. It shows that UAutomizer performs better on witnesses that are reduced with $\mathsf{r}_{\mathsf{state}}$ than with $\mathsf{r}_{\mathsf{all}}$: With $\mathsf{r}_{\mathsf{state}}$, UAutomizer improves its confirmation rate for all producers. It now only experiences a decrease of −10 % compared to the decrease of −48 % with $\mathsf{r}_{\mathsf{all}}$ (Fig. 7b). Additionally, the confirmation rate is now similar to the confirmation rate of the original witnesses for all producers but PeSCo [34, 35].

Our three reduction strategies have different advantages: $\mathsf{r}_{\mathsf{all}}$ shrinks violation witnesses significantly, while $\mathsf{r}_{\mathsf{state}}$ keeps the structure of the witness but removes superfluous assumptions. Strategy $\mathsf{r}_{\mathsf{match}}$ combines the advantages and disadvantages of the other two strategies.

4.3 Threats to Validity

Internal Validity. CPAchecker exports fault candidates in a JSON format and describes their location by providing the line number and a character offset in the input program. Verifiers export witnesses in a `GraphML` format and also describe file locations of source-code guards with four values, namely `startline`, `endline`, `startoffset`, and `endoffset`. These values should help to uniquely identify statements in the program. However, most witnesses only contain the key `startline` in their guards Therefore, matching the suspects to source-code guards can cause problems as it might be ambiguous.

To prevent the introduction of a bias to the existing witnesses, we apply the reduction directly to the original witnesses with our tool Flow instead of using the witness export of CPAchecker. Additionally, we evaluate our approach on 14 verifiers and 4 validators of SV-COMP 23 since all tools in SV-COMP support witnesses. A high number of verifiers and validators minimizes potential biases in our approach towards specific tools.

We re-implement MaxSat, an established fault localization technique in CPAchecker. The soundness of MaxSat was evaluated in the original publication [29]. We are confident that our implementation is sound as well. MinUnsat and Unsat are heavily inspired by MaxSat as seen in the similarities of the algorithms sketched in Algs. 1 and 2.

External Validity. We use the SV-COMP benchmark set for evaluating our approach. It is the largest available benchmarks set for verification tasks in the programming language C. Naturally, every benchmark set has a bias and we cannot be sure if the approach performs as well on other benchmark sets. However, this set is community maintained and everyone can contribute. It consists of a diverse task set.

To run our experiments, we use BenchExec, a tool for reliable benchmarking. Nonetheless, there is always the risk of imprecise measurements. Tasks near our timelimit of 90 s may be flaky. We implement the fault localization and reduction techniques strictly deterministic.

5 Conclusion

We presented three reduction strategies to reduce violation witnesses based on three different fault-localization approaches. An extensive evaluation on 21 356 original violation witnesses of the *Reach-Safety* category of SV-COMP 2023 showed that our reductions can significantly reduce the size of witnesses and increase their confirmation rate. This makes it easier to store and read them, as well as easier to handle them automatically.

Data-Availability Statement. The experiment setup and all experimental data are archived and available at Zenodo [11]. Our supplementary webpage at `https://www.sosy-lab.org/research/fl-witnesses` provides easy access to the data and additional information.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY), 418257054 (Coop), and 496588242 (IdeFix).

References

1. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C.: A practical evaluation of spectrum-based fault localization. J. Syst. Softw. **82**(11), 1780–1792 (2009). `https://doi.org/10.1016/j.jss.2009.06.035`
2. Ayaziová, P., Strejček, J.: Symbiotic-Witch 2: More efficient algorithm and witness refutation (competition contribution). In: Proc. TACAS (2). pp. 523–528. LNCS 13994, Springer (2023). `https://doi.org/10.1007/978-3-031-30820-8_30`
3. Beyer, D.: A data set of program invariants and error paths. In: Proc. MSR. pp. 111–115. IEEE (2019). `https://doi.org/10.1109/MSR.2019.00026`
4. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). `https://doi.org/10.1007/978-3-031-30820-8_29`
5. Beyer, D.: Verification witnesses from verification tools (SV-COMP 2023). Zenodo (2023). `https://doi.org/10.5281/zenodo.7627791`
6. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). `https://doi.org/10.1109/ICSE.2004.1317455`
7. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). `https://doi.org/10.1145/3477579`

8. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
9. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
10. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
11. Beyer, D., Kettl, M., Lemberger, T.: Reproduction package for article 'Fault localization on witnesses'. Zenodo (2024). https://doi.org/10.5281/zenodo.10952383
12. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
13. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
14. Beyer, D., Kettl, M., Lemberger, T.: Flow: Fault localization on witnesses. https://gitlab.com/sosy-lab/software/fault-localization-on-witnesses (2023), [Online; accessed 22-January-2024]
15. Beyer, D., Kettl, M., Lemberger, T.: Fault localization on verification witnesses (poster paper). In: Proc. ICSE. ACM (2024). https://doi.org/10.1145/3639478.3643099
16. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report. In: Graph Drawing. pp. 501–512. LNCS 2265, Springer (2001). https://doi.org/10.1007/3-540-45848-4_59
17. Chalupa, M., Henzinger, T.: Bubaak: Runtime monitoring of program verifiers (competition contribution). In: Proc. TACAS (2). pp. 535–540. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_32
18. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
19. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: Proc. FM. pp. 187–201. LNCS 7436, Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_17
20. Ernst, G., Blau, J., Murray, T.: Deductive verification via the debug adapter protocol. In: Proença, J., Paskevich, A. (eds.) Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021. EPTCS, vol. 338, pp. 89–96 (2021). https://doi.org/10.4204/EPTCS.338.11
21. Gadelha, M.Y., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. Int. J. Softw. Tools Technol. Transf. **19**(1), 97–114 (February 2017). https://doi.org/10.1007/s10009-015-0407-9
22. Gennari, J., Gurfinkel, A., Kahsai, T., Navas, J.A., Schwartz, E.J.: Executable counterexamples in software model checking. In: Proc. VSTTE. pp. 17–37. LNCS 11294, Springer (2018). https://doi.org/10.1007/978-3-030-03592-1_2
23. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Proc. SPIN. pp. 121–135. LNCS 2648, Springer (2003). https://doi.org/10.1007/3-540-44829-2_8
24. Heizmann, M., Barth, M., Dietsch, D., Fichtner, L., Hoenicke, J., Klumpp, D., Naouar, M., Schindler, T., Schüssele, F., Podelski, A.: Ultimate Automizer 2023 (competition contribution). In: Proc. TACAS (2). pp. 577–581. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_39

25. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
26. Jhala, R., Majumdar, R.: Path slicing. In: Proc. PLDI. pp. 38–47. ACM (2005). https://doi.org/10.1145/1065010.1065016
27. Jones, J.A., Harrold, M.J.: Empirical evaluation of the Tarantula automatic fault-localization technique. In: Proc. ASE. pp. 273–282. ACM (2005). https://doi.org/10.1145/1101908.1101949
28. Jose, M., Majumdar, R.: Bug-assist: Assisting fault localization in ANSI-C programs. In: Proc. CAV. pp. 504–509. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_40
29. Jose, M., Majumdar, R.: Cause clue clauses: Error localization using maximum satisfiability. In: Proc. PLDI. pp. 437–446. ACM (2011). https://doi.org/10.1145/1993498.1993550
30. Kölbl, M., Leue, S., Wies, T.: Tartar: A timed automata repair tool. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 529–540. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-53288-8_25
31. Metta, R., Yeduru, P., Karmarkar, H., Medicherla, R.K.: VeriFuzz 1.4: Checking for (non-)termination (competition contribution). In: Proc. TACAS (2). pp. 594–599. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_42
32. Monat, R., Ouadjaout, A., Miné, A.: Mopsa-C: Modular domains and relational abstract interpretation for C programs (competition contribution). In: Proc. TACAS (2). pp. 565–570. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_37
33. Müller, P., Ruskiewicz, J.N.: Using debuggers to understand failed verification attempts. In: Proc. FM. pp. 73–87. LNCS 6664, Springer (2011). https://doi.org/10.1007/978-3-642-21437-0_8
34. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
35. Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
36. Rockai, P., Barnat, J.: DivSIM, an interactive simulator for LLVM bitcode. STTT **24**(3), 493–510 (2022). https://doi.org/10.1007/s10009-022-00659-x
37. Saan, S., Schwarz, M., Erhard, J., Pietsch, M., Seidl, H., Tilscher, S., Vojdani, V.: Goblint: Autotuning thread-modular abstract interpretation (competition contribution). In: Proc. TACAS (2). pp. 547–552. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_34
38. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. IEEE Trans. Reliab. **63**(1), 290–308 (2014). https://doi.org/10.1109/TR.2013.2285319
39. Ádám, Z., Bajczi, L., Dobos-Kovács, M., Hajdu, A., Molnár, V.: Theta: Portfolio of cegar-based analyses with dynamic algorithm selection (competition contribution). In: Proc. TACAS (2). pp. 474–478. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_34