Testing in Formal Verification via Witness Generation (Empirical Evaluation)

Dirk Beyer, Thomas Lemberger, and Henrik Wachowitz

LMU Munich, Munich, Germany

Abstract. Despite potential synergies, the communities surrounding formal software verifiers and automatic test generators have developed different formats to describe a path to an error. Test generators export a test case whose execution makes the error observable, while verifiers produce a violation witness, an abstract description of the error path. Previous work transformed violation witnesses into test cases and evaluated their effectiveness, and other work found that test generators are more effective in bug finding than formal verifiers. While there are hybrid approaches to formal verification that utilize testing, there is no empirical evaluation of the usefulness of the off-the-shelf use of test generators in formal verification. We change that by transforming test cases to violation witnesses. This allows the use of test generators for finding counterexamples in verification scenarios like the Competition on Software Verification (SV-COMP), both directly and as parts of bigger verification systems. In a large empirical evaluation we examine the potential improvements this use of test generators can add to formal verifiers.

1 Introduction

Automated formal software verification and test generation are two complementary approaches to ensure software quality. Formal verification aims to either find program errors or prove their absence, while test generation focuses on finding error-triggering inputs. Despite potential synergies, the communities surrounding verifiers and test generators have developed different formats to describe a path to a found error. Test generators export a test case whose execution makes the error observable, while verifiers produce a *violation witness*, an abstract description of the error path. Previous work [1, 2] turned violation witnesses into test cases, and many other works [3, 4, 5, 6, 7, 8, 9, 10, 11] showed that the adaption of formal techniques can improve the effectiveness of test generation. By now, participants of Test-Comp use formal methods extensively [12]. But the application of test generation may be valuable for formal verification, as well: A previous study [12] on SV-COMP [13] and Test-Comp [14] indicates that current test generators are more effective in bug finding than formal verifiers. Still only two [11, 15] SV-COMP participants use dynamic approaches for automated software verification.

This paper closes the gap: We evaluate the off-the-shelf use of test generators in formal verification on the largest available benchmark set for software verification,

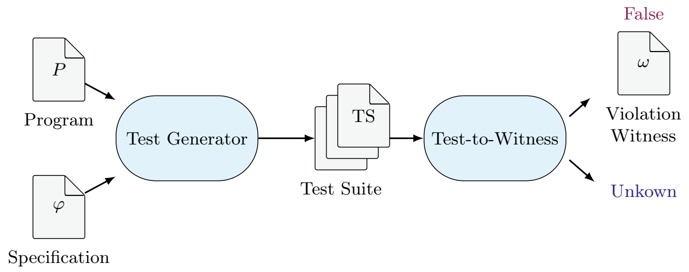


Fig. 1: Integrating test generators in the ecosystem of formal software verification

SV-Benchmarks [16], in the well-established ecosystem of SV-COMP [17]. To make this possible, test generators need to speak the same language as verifiers, i.e., provide violation witnesses instead of test cases. Our approach, Test-to-Witness, performs this transformation for test generators. Figure 1 shows the overall idea: We couple an off-the-shelf tester with Test-to-Witness to create, from the generated test suite, a violation witness; as long as the test suite contains at least one test case that triggers the error. This tool-chain can then be used in the ecosystem of formal software verification—we use it to evaluate potential gains.

We base Test-to-Witness on the uniform test-suite format of Test-Comp [18] to support a wide range of test generators. Unfortunately, no standalone fuzzer participated in the latest editions of Test-Comp [14, 18, 19, 20, 21, 22]. To include fuzzers in our evaluation, we introduce AFL-to-Test-Case (AFL-tc), which transforms the tests produced by AFL-fuzz [23, 24] into test cases in the Test-Comp format. From there we can apply Test-to-Witness. Our work answers the following research questions:

RQ 1 Can test cases be reliably transformed into violation witnesses?
Evaluation design: We use Test-to-Witness to generate violation witnesses from the existing test suites created in Test-Comp 2025.We check that the created witnesses can also be understood and confirmed by witness validators that participate in SV-COMP.

RQ 2 Does the standalone use of test generators yield competitive results in the falsification category of SV-COMP?
Evaluation design: The violation witnesses created in RQ 1 represent the results in error finding of Test-Comp 2025 participants in the SV-COMP setting. We directly compare this to the results of SV-COMP 2025.

RQ 3 What is the impact on overall efficiency if we combine test generators with formal verifiers?
Evaluation design: We compare the consumption of CPU time per verification task between the parallel portfolios and the corresponding standalone formal verifiers.

RQ 4 Does it improve the overall effectiveness if we combine test generators with formal verifiers?
Evaluation design: We create pairwise parallel-portfolio combinations of six test generators and six formal verifiers, and compare the number of solved tasks between the parallel portfolios and the standalone formal verifiers.

This extensive evaluation shows that the addition of test generators to formal verifiers can offer a significant increase in their bug-finding capabilities.

Related Work. Our work belongs into the ecosystem of automatic tools for verifying and testing software programs. A list of actively participating and well-maintained tools is available in the competition reports for SV-COMP [17] and Test-Comp [18]. Information and literature about most of the openly available verifiers and testers that participate in the competitions can be found online at https://fm-tools.sosy-lab.org/.

There are several surveys that describe the technology of tools for software testing [25, 26, 27, 28, 29, 30, 31] and software verification [32, 33, 34, 35]. Modern software testers and verifiers usually combine different techniques [36, 37, 38].

Another line of work is to use verification witnesses and test suites as exchange between different tools. For example, Witness2test takes as input a program and a violation witness and produces a test case that leads the program execution to the specification violation. The approach that we propose in this paper works in the opposite direction: It uses a program and a test case, runs the (instrumented) program and produces a witness that can be used for further analysis, such as visualization. There are several works concerned with explaining the results of software analysis [39, 40, 41, 42, 43, 44, 45].

2 Background

Program Representation. For the sake of presentation[1] we consider imperative, sequential programs over integers. A program $P = (L, \ell_0, E)$ consists of its program locations L, the initial program location $\ell_0 \in L$, and a set of program edges $E \subseteq L \times \mathrm{Ops} \times L$. Each program edge $(\ell, op, \ell') \in E$ represents a control-flow transition from program location ℓ to ℓ' by evaluating operation $op \in \mathrm{Ops}$. We consider three types of program operations: expression assignments, input assignments, and assumptions. An expression assignment $x \leftarrow expr$ assigns the value of expression $expr$ to program variable x. Expression $expr$ is an arithmetic expression over integers. An input assignment $x \leftarrow \circ$ receives an integer value from outside the program (e.g. sensor inputs, user inputs) and assigns it to program variable x. An assumption $[p]$ evaluates the Boolean expression p over program variables. Control-flow continues only if p evaluates to true. A program state $c : X \mapsto \mathbb{Z}$ is a mapping from variables to their assigned integer value.

An execution $[\![P]\!] = (\ell_0, c_0) \xrightarrow{op_0} (\ell_1, c_1) \ldots \xrightarrow{op_{m-1}} (\ell_m, c_m)$ of program P is a sequence where each step $(\ell_{i-1}, c_{i-1}) \xrightarrow{op_{i-1}} (\ell_i, c_i)$ corresponds to a program edge $(\ell_{i-1}, op_{i-1}, \ell_i) \in E$, and the program state c_i is the result of applying operation op_{i-1} to program state c_{i-1}.

Test Case. A test case $\boldsymbol{t} = \langle v_0, \ldots, v_{n-1} \rangle$ is a vector of input values v_i. The length n of a test case $\boldsymbol{t} = \langle v_0, \ldots, v_{n-1} \rangle$ is its number of input values. During program execution, whenever an input assignment $x \leftarrow \circ$ is evaluated, variable x is assigned the next input value from the test case.

[1] Our implementation works on C programs.

```
<testcase>
  <input>1</input>
  <input>0</input>
  <input>0</input>
</testcase>
```

Fig. 2: Test case in XML format that describes the test vector $\langle 1, 0, 0 \rangle$

```
- segment:                            - segment:
  - waypoint:                           - waypoint:
      action: "follow"                      action: "follow"
      location:                             location:
        file_name: "prog.c"                   file_name: "prog.c"
        line: 5                               line: 14
        column: 35                            column: 5
      type: "function_return"               type: "target"
      constraint:
        format: "acsl_expression"
        value: "\\result == 0"
```

Fig. 3: Segments of a violation witness

Test-Comp requires participants to produce test cases in XML format. Each test case is a separate XML file with a sequence of `<input>` elements. Figure 2 shows the test case in XML format for test vector $\langle 1, 0, 0 \rangle$. A Test-Comp test suite is a collection of test cases together with a metadata file. A test suite is *successful* if it contains at least one test case that induces a program execution that reaches a call to `reach_error()`.

Competitions. Competitions are a scientific method for comparative evaluation, in which participants submit their tools together with instructions how to execute them, and an organizer executes all experiments under the same conditions. The competition on software verification SV-COMP annually evaluates verifiers, which take as input a program and a safety or liveness specification and produce as output a result *true* (together with a correctness witness), `false` (together with a violation witness), or *unknown* (if the verifier cannot determine the result). The competition on software testing Test-Comp annually evaluates testers, which take as input a program and a coverage specification and produce as output a test suite. SV-COMP validates the quality of a reported verification result through witness validation: A verifier only receives points for a correct verification result if it also produced a witness that can be validated.

Both SV-COMP and Test-Comp use the SV-Benchmarks collection of verification and test tasks, which is the largest and most diverse benchmark set of testing and verification tasks for the languages C and Java. The benchmark set contains verification tasks for properties such as reachability, memory safety, termination, no-overflows, and no-data-races, which are used by SV-COMP. Test-Comp uses tasks with coverage specification `cover-branches` and `cover-error`. Tasks with coverage specification `cover-error` must fulfill the following conditions: (a) the program contains at least one call to an input method `__VERIFIER_nondet_X`, (b) the program always terminates, (c) the program compiles successfully, and (d) the program has a reachability property with expected verdict `false` (i.e., it can reach a call to function `reach_error`). In consequence, the tasks that are used in Test-Comp with coverage specification `cover-error` are a subset of the tasks that are used in SV-COMP for reachability verification.

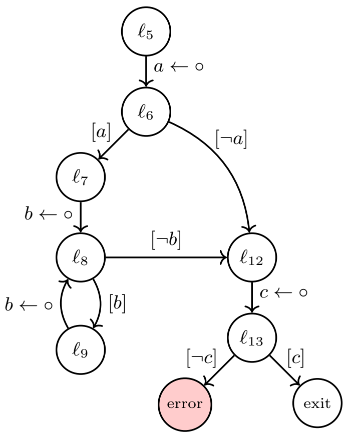
Violation Witnesses 2.0. A decade ago, violation witnesses [46] were introduced in the area of software verification and adopted by SV-COMP. Later, witnesses for correctness [47], non-termination, and other properties were introduced. While the first generation of witnesses were based on an XML format, the more recent witnesses in version 2.0 [48] are based on the YAML format. In this paper we use violation witnesses in version 2.0. Their aim is to support validation of the verification verdict `false`; that is, there is a path through the program that violates a safety specification.

A violation witness describes a set of program executions, of which at least one must reach a violation of the specification. The set of executions is described as a sequence of segments (cf. Fig. 3). Each segment consists of at least one waypoint `follow` and multiple optional waypoints `avoid`. Each waypoint is anchored to a program location (line number and column), and can hold assumptions over program variables, describe branching decisions or function-return values, or signal a function entry. The last segment of a violation witness contains a waypoint `target`, which indicates that the violation of the specification is reached. Figure 3 shows two example segments: The first segment contains a waypoint `follow` of type `function_return` that matches some function call at line 15, column 33 of file `prog.c`. This function call's constraint `\\result == 0` expresses that the call's return value must be 0 for the path to be valid. The second segment contains a waypoint `follow` of type `target`. This claims that the path should reach line 8, column 5 of file `prog.c`, where a specification violation occurs.

A validator for violation witnesses can re-play the described paths to show that one of the paths is feasible and indeed violates the specification. For a described path to be feasible, there must be a program execution that passes all waypoints, satisfies all assumptions along the way, and reaches the target. There is broad support for witness format version 2.0, but only two validators in SV-COMP 2025 support function-return assumptions: CPAchecker and Witch.

3 Test-to-Witness

Consider the example program in Fig. 4, and two test cases that both trigger the error at ℓ_8: $\mathrm{tc}_1 = \langle 0, 0\rangle$ and $\mathrm{tc}_2 = \langle 1, 0, 0\rangle$. Program execution with test case tc_1 introduces the first input value 0 at ℓ_6 and the second input value 0 at ℓ_{12}, while program execution with test case tc_2 introduces the first input value 1 at ℓ_6, but then assigns the second input value at ℓ_7 before the third input value 0 is assigned at ℓ_{12}. This shows the key challenge in converting a test case to a violation witness: the test case lists the sequence of input values to give to a program execution, but it has no information on *where* in the program an input value is used. This information is strictly necessary to describe waypoints in witnesses. Because the control flow of two program executions can lead to a different order of calls to input methods, pattern matching on the program code is not sufficient. Instead, it is necessary to consider the program semantics to reliably match which input method consumes which input value. Input methods within loops (as induced by ℓ_9) may also create test cases of dynamic length.



```c
extern int __VERIFIER_nondet_int(void);
extern char __VERIFIER_nondet_char(void);
void reach_error() {/* .. */};
int main() {
  char a = __VERIFIER_nondet_char();
  if (a) {
    int b = __VERIFIER_nondet_int();
    while (b) {
      b = __VERIFIER_nondet_int();
    }
  }
  int c = __VERIFIER_nondet_int();
  if (!c) {
    reach_error();
  }
}
```

Fig. 4: Program in graph representation (left) and corresponding C code (right)

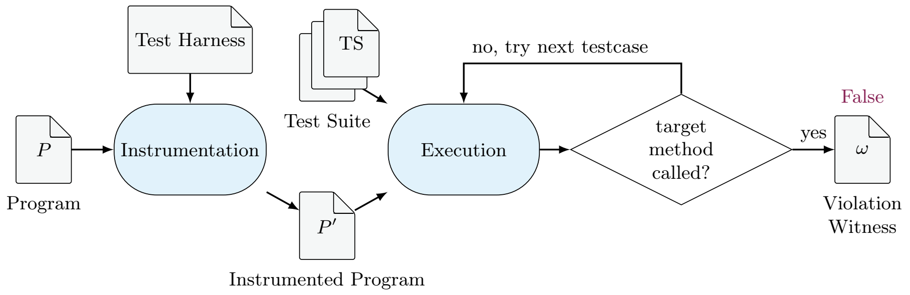


Fig. 5: Workflow of Test-to-Witness

Test-to-Witness solves this issue through a projection of the program execution trace: Given a program P and a test case $\boldsymbol{t} = \langle v_0, \ldots, v_{n-1} \rangle$, Test-to-Witness executes P with $\boldsymbol{t}$ to obtain a finite execution trace $[\![P]\!]_{\boldsymbol{t}} = (\ell_0, c_0) \xrightarrow{\mathrm{op}_0} (\ell_1, c_1) \ldots$ for $\boldsymbol{t}$. To receive a violation witness from this trace, Test-to-Witness projects $[\![P]\!]_{\boldsymbol{t}}$ onto input operations $\mathrm{op}_j = w_j \leftarrow \circ$ and the valuation $c_j(w_j)$:

$$vs = \langle (\ell_{i_1}, c_{i_1}(w_{i_1})), (\ell_{i_2}, c_{i_2}(w_{i_2})), \ldots, (\ell_{i_k}, c_{i_k}(w_{i_k})) \rangle$$

where $\{i_1, i_2, \ldots, i_k\} = \{j \mid \mathrm{op}_{j-1} = w_j \leftarrow \circ\}$ and $i_1 < i_2 < \ldots < i_k$. Sequence vs describes that at program location ℓ_{i_j}, variable w_{i_j} is assigned the test input value $c_{i_j}(w_{i_j}) = \boldsymbol{t}_{i_j}$. Each tuple in vs is then translated to its corresponding segment in the violation witness, specifying the program location through source-code line and column, and the input value to assume as constraint. A formalization of the segments can be found in the SV-COMP witness format specification [49].

Our implementation of Test-to-Witness works on C programs. It realizes the projection through a program instrumentation that, when executed with a test

```
1  extern int __VERIFIER_nondet_int();
2  extern char __VERIFIER_nondet_char();
3  void reach_error() {/* .. */};
4  int __VERIFIER_nondet_int_log(int line, int column) {
5    int val = __VERIFIER_nondet_int();
6    /* .. print violation-witness segment for line, column,
7       with assumption '\result = {val}' */
8    return val;
9  }
10 char __VERIFIER_nondet_char_log(int line, int column) {
11   char val = __VERIFIER_nondet_char();
12   /* .. print violation-witness segment for line, column,
13      with assumption '\result = {val}' */
14   return val;
15 }
16 void reach_error_log(int line, int column) {
17   /* .. print violation-witness segment with target waypoint
18         for line, column */
19   reach_error();
20 }
21 int main() {
22   char a = __VERIFIER_nondet_char_log(5, 35);
23   if (a) {
24     int b = __VERIFIER_nondet_int_log(7, 35);
25     while (b) {
26       b = __VERIFIER_nondet_int_log(9, 33);
27     }
28   }
29   int c = __VERIFIER_nondet_int_log(12, 33);
30   if (!c)
31   {
32     reach_error_log(14, 5);
33   }
34 }
```

Fig. 6: Instrumented version of `control-flow-example.c`

case, directly writes a violation witness with the correct location information. Figure 5 shows this workflow. Test-to-Witness first instruments the program P and compiles it against a test harness. To represent input assignments, it relies on the SV-COMP convention of using input methods `__VERIFIER_nondet_X`, where X is the type of the returned value. It considers calls to method `reach_error(void)` as specification violations. This property can be reduced to other reachability properties, like failing assertions.

After compilation, Test-to-Witness executes the resulting binary with all test cases of the test suite until one test case reaches the target method. If this happens, the violation witness produced by this execution is returned.

Program Instrumentation. For each input method, Test-to-Witness defines a new variant that (1) receives its call location in the program file as two parameters `line` and `column`, (2) calls the original input method and stores the returned value in a local variable `val`, (3) prints a fragment of the violation witness that describes that the return value of the input method must be equal to `val`, and (4) returns `val` to the caller. For each target method, Test-to-Witness defines a variant that first prints the fragment of the violation witness and then calls the original target method. Figure 6 shows these method definitions for Fig. 4 in lines 4–34. Test-to-Witness then replaces all calls to input methods and target methods with the new variants, passing the appropriate line and column numbers

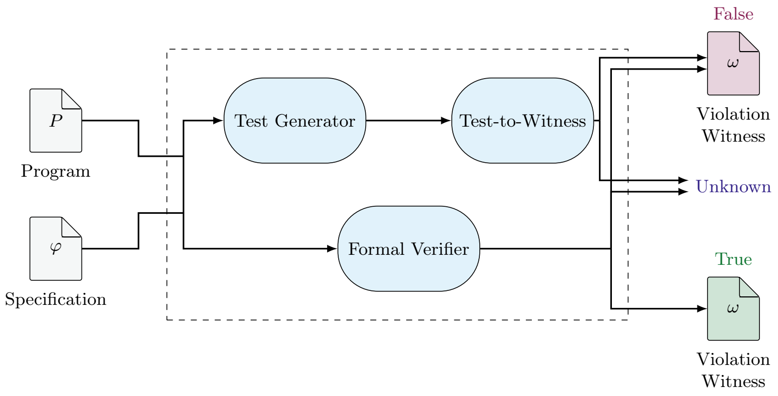


Fig. 7: Architecture of our Parallel Portfolio

for each original call site. To match the specific waypoint of violation witnesses, the column number is the position of the closing parenthesis of the original method call for input methods, and the position of the first character of the method call for target methods. Last, Test-to-Witness creates a test harness that defines the original input methods so that they parse input values from standard input, and compiles the instrumented program against this harness. Because the resulting binary parses input values from standard input, the same binary can be used to execute all available test cases.

Execution with Test Cases. Test-to-Witness executes the compiled binary with each test case of the test suite until one execution reaches the target method. To not be blocked by slow test executions, Test-to-Witness executes multiple test cases in parallel. Each execution's produced violation witness is captured to a separate file. As soon as one execution reaches the target method, Test-to-Witness terminates all other executions and returns the corresponding violation witness.

Testers as Falsifiers. Figure 1 shows how any off-the-shelf tester can be used with Test-to-Witness to produce violation witnesses for formal verification. Similar to a formal verifier, the tester receives the program P under analysis and a specification φ.[2] Once started, the tester tries to find a test case that violates the specification. Testers consider the given specification to various degrees: Some testers stop as soon as a test case is found that matches the test-goal specification [50, 51], while others generate test cases for internally implemented coverage criteria [52], or fully randomly [23, 53]. For the latter we can monitor the generated test cases continuously and still stop the tool as soon as one covering test case is found.

Parallel Portfolio. Testers can only find specification violations, but not proof the absence of violations. Because of this, the use of testers in formal verification is mostly useful in combination with a formal verifier. For this we propose a

[2] We do not show a possible translation of a verification specification to a test goal specification, as this is not the focus of this paper. In our context this was merely matching existing verification specifications to test goals.

parallel-portfolio approach (Fig. 7), where a tester runs concurrently to a formal verifier. Whenever the tester produces a test case, Test-to-Witness tries to convert the test case into a witness. If the conversion is successful, the portfolio terminates and returns the witness generated by Test-to-Witness. If the conversion fails, the portfolio continues. The two most common reasons for failure are: (1) the tester was aborted before it could find a violating test case and (2) the test case does not lead to a violation of the property; it is a false positive. We implement the parallel portfolio in the cooperative verification framework CoVeriTeam [54]. CoVeriTeam provides unified interfaces for both testers and formal verifiers, and allows to specify the specific tools to use through the command line. This enables us to plug-and-play different combinations of testers and verifiers easily.

AFL-TC. To enable the use of AFL-FUZZ with Test-to-Witness, we implement AFL-TC, a tool chain that runs AFL-FUZZ in parallel with a monitor process. It continuously monitors the test cases generated by AFL-FUZZ. Whenever a test case is produced that triggers a program crash, the monitor invokes Test-to-Witness to transform the test case to a violation witness. If this transformation is successful, the monitor terminates AFL-FUZZ and returns the violation witness.

4 Evaluation

We answer the following research questions with an experimental evaluation:

RQ 1 Transformation: Can test cases be reliably transformed into violation witnesses?
RQ 2 Viability: Are the standalone use of Test-Comp test generators competitive in the falsification category of SV-COMP?
RQ 3 Efficiency: Does the combination of Test-Comp test generators with verifiers improve the overall efficiency?
RQ 4 Effectiveness: Does the combination of Test-Comp test generators with verifiers improve the overall effectiveness?

Tool Versions. We use verification tasks from SV-Benchmarks [55] in the version used for SV-COMP 2025 [17]. We use Test-to-Witness revision c9d2a32. For the portfolio, we use CoVeriTeam [54] revision 0408cbd. For reliable resource measurement, we use BENCHEXEC [56] b07c314d. All experiments were executed on machines equipped with one Intel Xeon E3-1230 v5 (3.4 GHz, 8 processing units) and 33 GB of RAM, running Ubuntu 24.04 LTS 64 bit and GNU/Linux 6.8.0.

RQ 1: Can test cases be reliably transformed into violation witnesses?
We let Test-to-Witness transform test suites that were produced in Test-Comp 2025 into violation witnesses. We consider each test suite that was produced in the `cover-error` category and that contains at least one test case where TESTCOV [57] confirmed in Test-Comp that its execution calls the error function.

This selection consists of 10 935 test suites that were produced by 18 testers. We run Test-to-Witness on all test suites, with a timeout of 5 min per test suite.

Table 1: Summary of test cases that were transformed to violation witnesses

Total Tasks	Converted	Avg. Time (s)	Median Time (s)
10 935	10 897	0.77	0.16

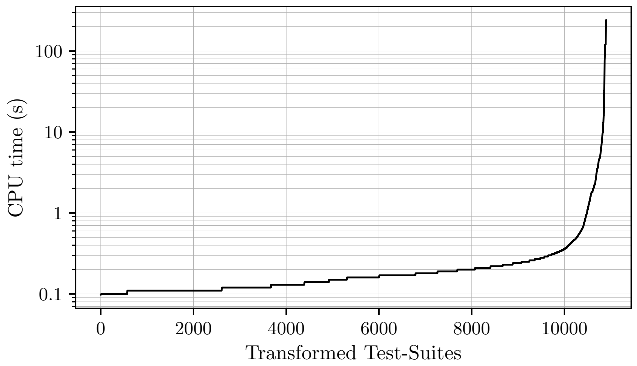


Fig. 8: Quantile plot of transformation times for test cases to violation witnesses

Table 1 shows that 10897 ($\approx 99.6\,\%$) test suites were transformed successfully within the given time limit. The quantile plot in Fig. 8 shows the distribution of transformation times that Test-to-Witness requires to transform the test suites into violation witnesses. The x-axis shows the number of test suites that can be transformed within the CPU time that is given on the y-axis. For example, the graph shows that 8 069 test suites ($\approx 74\,\%$) are transformed by Test-to-Witness with a run time of 0.2 s or less, and 10 490 test suites ($\approx 96\,\%$) are transformed within 1 s or less. This shows that the time required to transform a test suite into a violation witness is usually so low that it is negligible.

Next we explore the impact that the size of a test suite has on the transformation time. Figure 9 shows the distribution of test-suite sizes on the left plot. The x-axis shows the size of the test suite and the y-axis is the number of test suites that have at least x amount of test cases. The median size of a test suite is 1, but we also observe significant outliers.

The scatter plot on the right side of Fig. 9 shows the effect of test-suite size on transformation time. The x-axis shows the size of a test suite and the y-axis shows the CPU time that the transformation takes. We observe a trend that the transformation of larger test suites consumes more CPU time. We expect this, since Test-to-Witness executes test cases in parallel until one is found that reaches the error location. We look at two example test suites whose transformation takes long: The test suite created by TracerX-WP for the task `list-2.yml` takes 78 s to transform. The suite consists of 20 774 test cases. Each individual test case only takes a few milliseconds to execute, but the cumulative time for all test cases quickly adds up, especially since Test-to-Witness runs as many test-cases in parallel as there are CPU cores available.

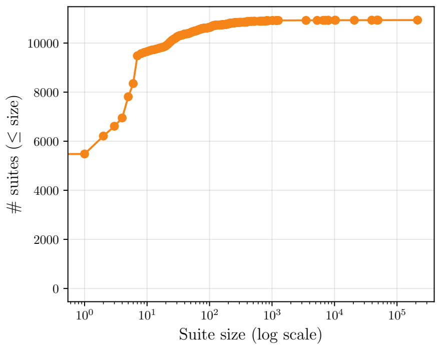


Fig. 9: Distribution of test-suite sizes

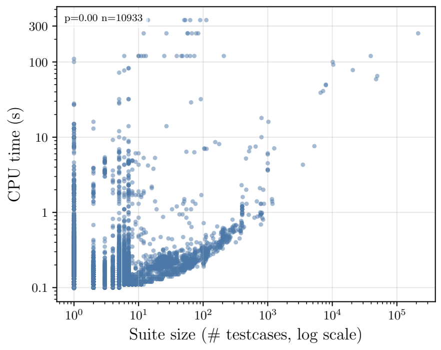


Fig. 10: Effect of test-suite size on transformation time (in CPU time)

But also test suites with few test cases can take a long time to transform: The test suite created by Kleef for task `btor2c-lazyMod.unsafe_buggy_ridecore.yml` consists of a single test case, but takes 110 s to transform. The compilation of the instrumented program dominates this transformation time. Executing the test case itself takes less than 1 s.

We evaluate the quality of the generated violation witnesses by running the two SV-COMP validators on them that support function-return constraints: CPAchecker and Witch. We run both validators with a timeout of 90 s per witness, which is the limit for violation-witness validation in SV-COMP.

Table 2 shows, for each test generator, the number of test suites created by that test generator (# Suites), the number of witnesses that are successfully transformed into violation witnesses (# Converted), the number of witnesses that are confirmed by at least one validator (# Confirmed), and the percentage of considered test suites that are successfully transformed and confirmed (% Confirmed). In total, 9 739 (≈ 89 %) of the 10 935 test suites are successfully transformed and subsequently confirmed by a validator.

From these results we conclude that test cases can be reliably transformed into violation witnesses. Almost all test suites can be transformed within 1 s of CPU time, and almost 90 % of the generated witnesses are confirmed by at least one of the two validators. The extremely efficient transformation makes the approach feasible for real-time integration into test-to-verification-workflows such as the portfolio presented in Sect. 3.

RQ 2: Does the standalone use of Test-Comp test generators yield competitive results in the falsification category of SV-COMP?

Since the benchmark set of Test-Comp is a subset of the benchmark set of SV-COMP, we can compare the results of test generators in Test-Comp to the results of verifiers in SV-COMP. We achieve this by comparing the *validated* test-generator results from Test-Comp 2025 to the verifier results of SV-COMP 2025. We define a *validated* test-generator result as a test suite that was successfully transformed into violation witnesses and then confirmed by at least one validator—this is the criterion that a *tester as falsifier* would need to fulfill to score points in SV-

Table 2: Transformed and validated violation witnesses from test cases

Tester	# Suites	# Converted	# Confirmed	% Confirmed
cetfuzz	320	317	313	98
esbmc-incr	948	948	772	81
esbmc-kind	948	948	770	81
fdse	628	626	589	94
fizzer	612	608	589	96
fusebmc	979	978	826	84
fusebmc-ia	951	950	844	89
hybridtiger	428	423	421	98
klee	889	887	759	85
kleef	998	995	841	84
owic	230	227	227	99
prtest	295	290	240	81
rizzer	550	549	503	92
symbiotic	586	584	581	99
tracerx	420	419	391	93
tracerx-wp	402	400	360	90
utestgen	317	314	311	98
wasp-c	434	434	402	93
Total	10 935	10 897	9 739	89

COMP. To enable a fair comparison, we add the CPU time that Test-to-Witness takes to transform the test suites into violation witnesses to the time the test generator initially needed to generate the test suite in Test-Comp 2025. We restrict ourselves to the `ReachSafety` category of tasks, where we only consider tasks with an expected verdict of `false`; since test generators cannot find proofs.

On the entire benchmark set of tasks with an expected verdict of `false`, test generators are not competitive with verifiers. The best verifier solves 1 943 tasks, while the best test generator only solves 848 tasks. We discussed in Sect. 2 that not all tasks with an expected verdict of `false` qualify for test generation. The `ReachSafety` category of SV-COMP 2025 contains 2 929 tasks with an expected verdict of `false`. But out of those, Test-Comp 2025 restricts its benchmark set to the 1 074 tasks that qualify for test generation. We consider the results of verifiers and test generators on this subset. Figure 11 shows the quantile plot of correctly found alarms per tester and verifier. We highlight some verifiers and test generators: CPAchecker and Symbiotic as the two best performing verifiers in the `Falsification` category, and UAutomizer as the overall winner of SV-COMP 2025; FuSeBMC and Kleef as the two best performing test generators in the `cover-error` category of Test-Comp 2025. The quantile plot shows that these test generators outperform the verifiers. FuSeBMC, the winner of the `cover-error` category, is able to find 832 alarms within the time limit of 900 s, while the winning verifier of the Falsification and `ReachSafety` category, CPAchecker, finds only 706 alarms within the time limit. To get a better understanding of the potential improvements that are possible through tool combinations, we include the virtual

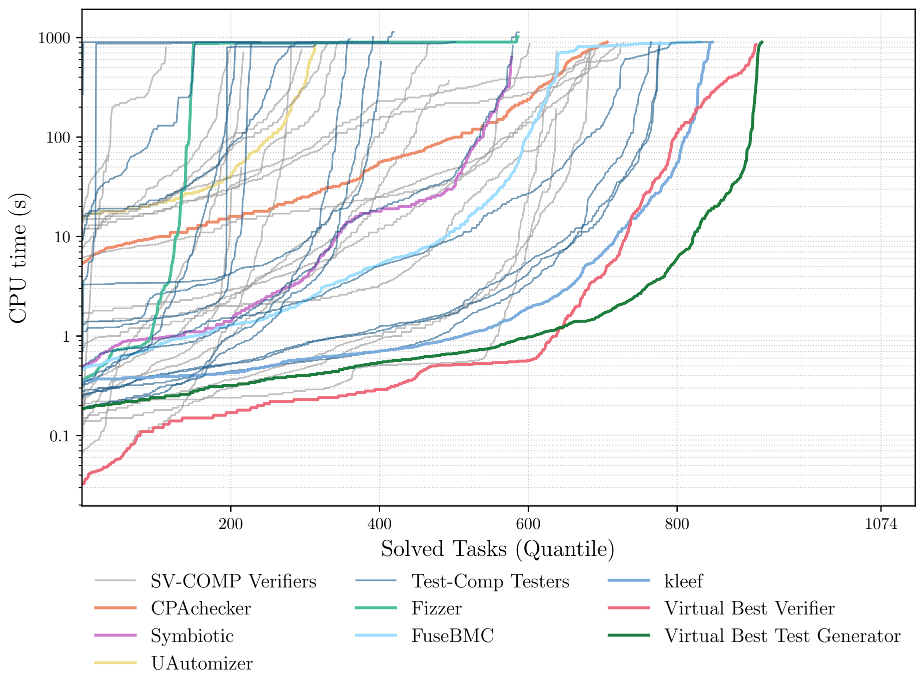


Fig. 11: Quantile plot of correctly found alarms per tester and verifier, limited to the 1 074 tasks that are suitable for test generation

best verifier and the virtual best test generator in Fig. 11. The virtual best verifier picks from all verifiers the fastest solving time per task; the virtual best test generator picks from all test generators the fastest time per task.

The results show that, if we level the playing field by restricting the benchmark set to tasks that are suitable for test generators, test generators yield competitive results in the falsification category of SV-COMP. When we create a virtual best tester and a virtual best solver, we observe only a thin advantage of test generators over verifiers.

RQ 3: Does the combination of test generators with verifiers improve the overall efficiency? We study the impact that combining test generators with formal verifiers has on the overall efficiency of verification. For this, we create a set of portfolio solvers that consist of one test generator and one verifier each, as described in Sect. 3.

We select a representative set of verifiers based on their performance in SV-COMP 2025: We select CPAchecker as the best tool in the `Falsification` and `ReachSafety` categories, Symbiotic, and Bubaak as the runner ups in the `Falsification` category, as well as CPV, and Esbmc-kInd as the runner ups in the `ReachSafety` category. Additionally we select UAutomizer as it is the overall winner of SV-COMP 2025. We select representative test generators based on their performance in Test-Comp 2025: We select FuSeBMC, Kleef, and Symbiotic as the top three test generators of the `cover-error` category. We also select Fizzer as the third place in the overall ranking of Test-Comp 2025. Finally, we select PRTest as a simple baseline tester that uses black-box random testing.

We run all combinations on the `ReachSafety` tasks contained in the SV-Benchmarks repository [16]. We limit each portfolio to a CPU time limit of 900 s, a memory limit of 15 GB and 4 CPU cores. The scatter plots in Fig. 12 show the CPU time consumption for tasks that both the combination of verifier and test generator as well as the reference alone solved correctly. The figure is structured in rows and columns, where each row represents a verifier and each column a test generator. For space reasons, we omit the test generators Symbiotic and Fizzer from the plots. The full plot is archived on Zenodo: 10.5281/zenodo.18312562. Each subplot has the same structure: the x-axis shows the CPU time needed by the reference verifier alone, while the y-axis shows the CPU time needed by the portfolio combination of verifier and test generator. Points below the diagonal line indicate that the portfolio combination was faster than the verifier alone, while points above the diagonal line indicate that the verifier alone was faster. The subplots also contain the number of considered tasks (N), and the P-Value of a Wilcoxon signed-rank test (p) [58], testing the one-sided hypothesis that the portfolio is slower than the reference. Orange dots indicate tasks that the reference was not able to solve, but that the portfolio was able to solve. Green dots indicate tasks that the portfolio was not able to solve, but that the reference was able to solve. Both orange and green dots are ignored for the statistical test. We only consider tasks that both variants solved correctly.

The addition of test generators does not improve the efficiency of verification, overall. In all considered combinations, the p-value is 0, indicating that the portfolios use significantly more CPU time than the verifiers alone. However, we can see an overall trend that the increase in CPU time is moderate. There are few outliers where the portfolio is significantly slower than the verifier alone. But we can still see improvements in the efficiency on individual tasks: fuzzing-based test generators such as AFL-TC can find violations quickly for some tasks, showing potential, e.g., in the combination with UAutomizer, CPAchecker, and CPV. The portfolios with PRTest also show potential: they have faster tasks along the 10 s line of the y-axis. This speeds up multiple tasks for every verifier, that originally takes hundreds of seconds to solve (points far right on the x-axis).

In summary, the addition of test generators to verifiers does not improve the overall efficiency. However, the increase in CPU time is moderate and does not constitute a practical limitation: in real-world usage, a moderate overhead in CPU time is acceptable when it leads to finding more bugs.

RQ 4: Does the combination of test generators with verifiers improve the overall effectiveness?

To check whether the addition of test generators to verifiers can increase the overall effectiveness, we analyze whether the combination can solve more tasks than the verifier alone. We use the same portfolio combinations as in RQ 3.

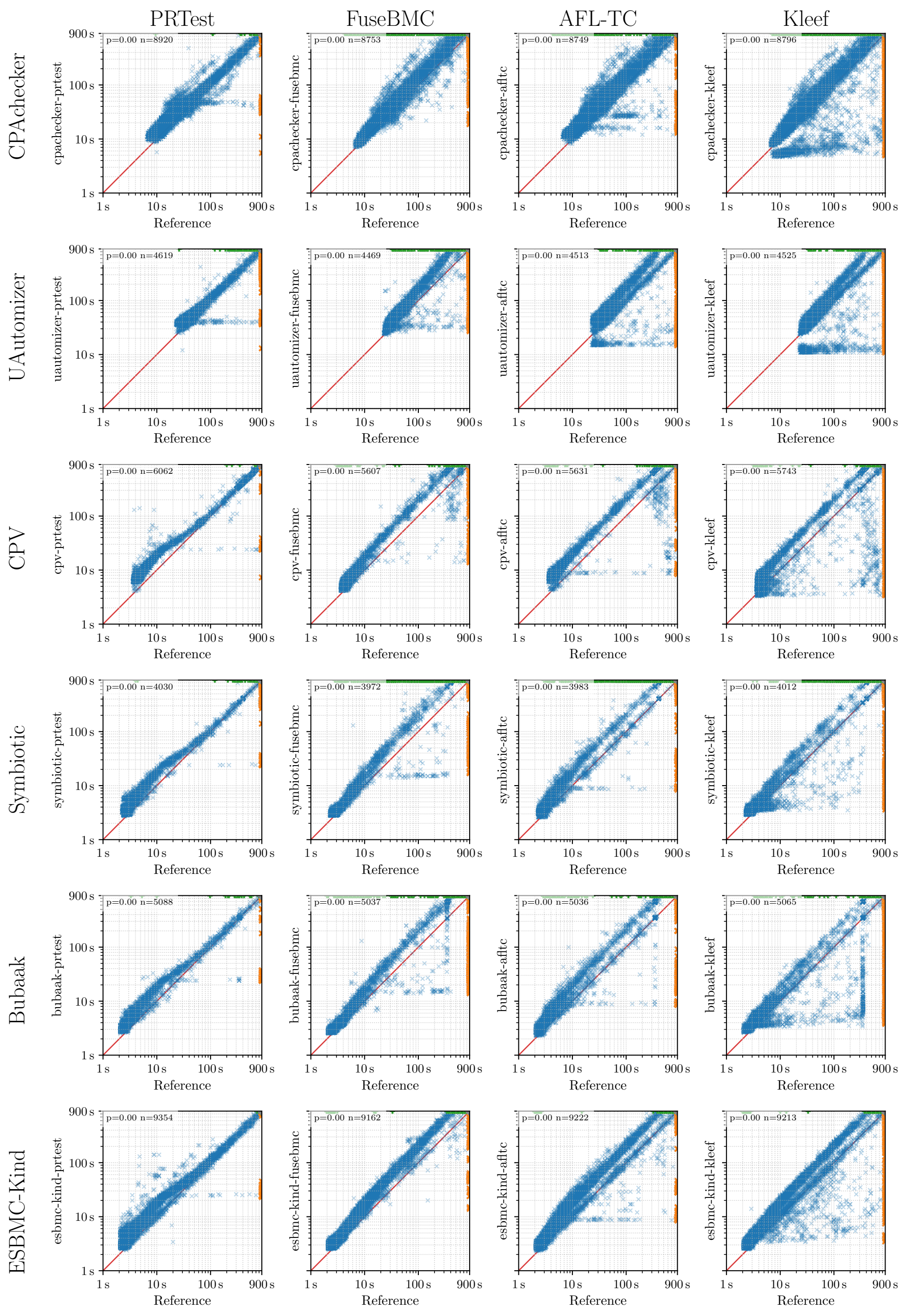


Fig. 12: Scatter plot of different verifier-test-generator-combinations

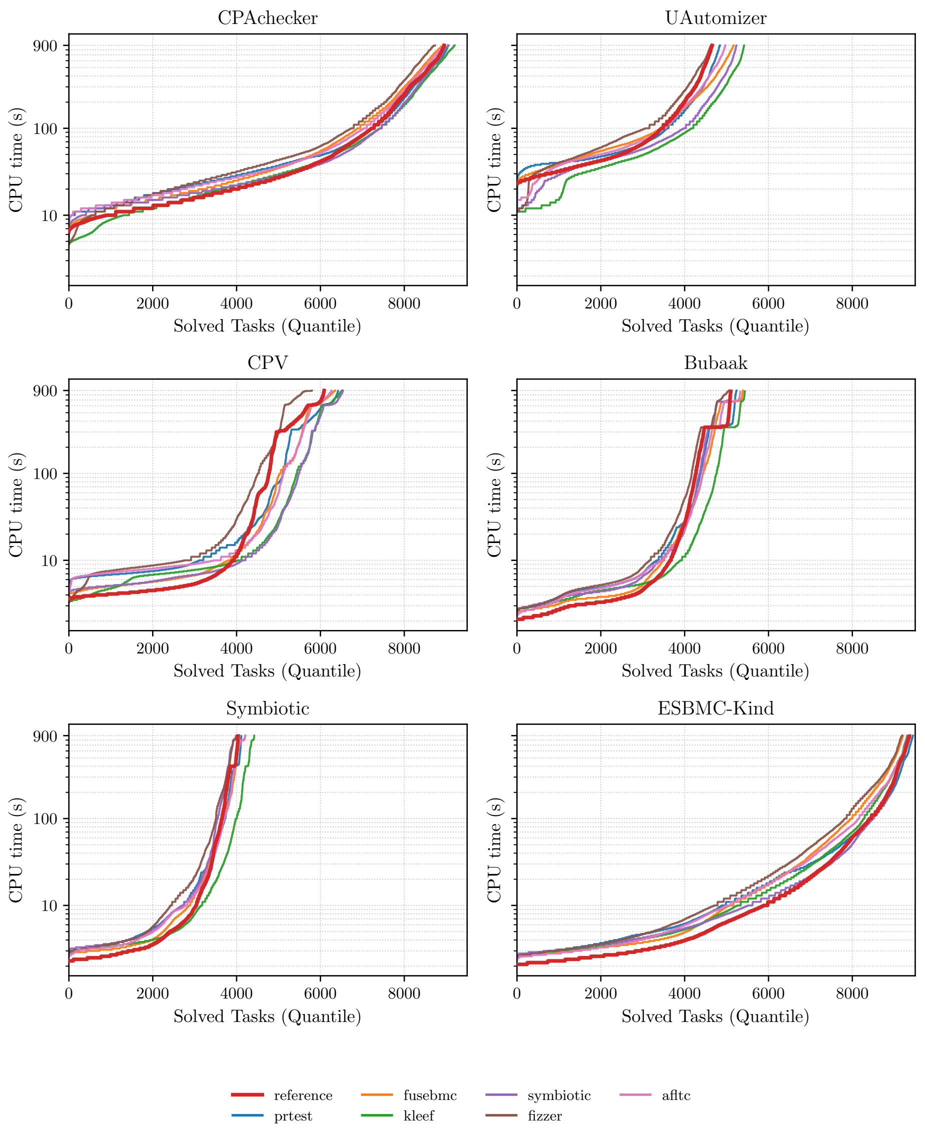


Fig. 13: Quantile plot of different verifier-test-generator combinations (number of solved tasks per configuration, out of 15 310 total tasks)

Table 3: Summary of gains and losses of verifier-test-generator-combinations; Uniquely solved (Unq) tasks are tasks that only the portfolio solved, not the reference; Timeouts (TO) and memory outs (MO) are tasks that the reference solved, but the portfolio did not due to timeout or memory out.

Verifier	PRTest			Fizzer			FuSeBMC			AFL-TC			Symbiotic			Kleef		
	Unq	TO	MO	Unq	TO	MO	Unq	TO	MO	Unq	TO	MO	Unq	TO	MO	Unq	TO	MO
CPAchecker	125	33	0	114	331	1	166	198	2	214	204	0	298	99	92	405	142	15
UAutomizer	222	21	12	153	149	40	702	172	18	457	122	23	679	27	75	893	84	48
CPV	360	6	0	372	643	0	746	460	1	631	458	0	770	56	244	774	321	3
Symbiotic	82	1	3	92	132	0	233	64	2	222	54	0	33	23	57	410	21	5
Bubaak	152	9	1	100	144	0	353	65	0	291	65	0	223	20	189	372	29	5
Esbmc-kInd	93	8	0	69	248	0	48	197	4	133	140	0	140	41	144	95	148	2

Figure 13 shows the results of all portfolio combinations. For each portfolio we also provide as reference a run of the verifier on its own, giving it the full time limit and memory limit. We observe that verifiers that excel in proving the absence of errors, such as UAutomizer and CPV, benefit more from the addition of test generators in the portfolio than verifiers that are already proficient in bug-finding (according to SV-COMP results), like CPAchecker and Esbmc-kInd. The portfolio combinations can, in some cases, end up solving less tasks than the verifier alone, indicated by the plot line ending further to the left than the reference line. But each verifier, when paired with an appropriate test generator, can improve their effectiveness, even when staying within the same CPU-time and memory limits given by SV-COMP.

We take a closer look at the gains and losses of the portfolio combinations in Table 3. One observation is that even adding plain random testing (PRTest) yields new uniquely solved tasks for all verifiers. The portfolios with more elaborate test generators have more potential for uniquely solved tasks, but also incur more losses due to timeouts and out-of-memory errors. These timeouts occur because in the CPU time limit is shared between the two tools in the portfolio, leaving less CPU time for each individual tool. In a practical setting, one could counteract these timeouts by increasing the overall CPU time limit of the portfolio. We also observe that most verifiers are not inhibited by out-of-memory errors when combined with test generators. Only UAutomizer suffers from a significant amount of out-of-memory errors when combined with test generators.

Another observation from Table 3 is that static-analysis based test generators such as Kleef and Symbiotic also yield new uniquely solved tasks for most verifiers—tools which are static analyzers themselves. As an example, both Kleef and CPAchecker implement a form of symbolic execution, yet their combination is able to solve 405 more tasks than CPAchecker alone. Even the portfolio of Symbiotic in verifier configuration with Symbiotic in test generator configuration is able to solve new tasks when compared to just Symbiotic (verifier) alone.

From the results we conclude that the addition of test generators to verifiers can significantly improve the overall effectiveness. All considered verifiers are able to solve more tasks when combined with an appropriate test generator. Most tasks that are not solved by the portfolio are due to timeouts, which can be counteracted by increasing the overall CPU time limit of the portfolio.

Threats to Validity. The validity of drawn conclusions is always limited: We consider the internal, external, and construct validity of the presented results.

Internal Validity. We evaluate combinations of tools rather than combinations of techniques in isolation, which means that observed effects may be due to specific implementations in the tools rather than the underlying techniques themselves. Many test generators use hybrid approaches [12], which prevents us from drawing conclusions about which specific techniques work well together. However, to minimize internal validity threats, we conduct our experiments on the same infrastructure as the official SV-COMP and Test-Comp competitions, ensuring consistency with the competition setting. We use BenchExec [56] for reliable resource measurements, which minimizes measurement errors and variability.

External Validity. Our evaluation is limited to the C programming language and the benchmark sets from SV-COMP and Test-Comp. However, these represent the largest publicly available benchmark set of verification tasks for C programs. Both competitions are highly concerned with precise measurements and reproducibility, lending credibility to our results. The tools we selected are proven to be good representatives of their kind, as they perform well in the respective competitions. Thus, while results may not generalize to other programming languages or entirely different domains, our findings are representative of the state-of-the-art in formal verification and testing for C.

Construct Validity. We design our experiments to evaluate whether test generators can improve verification. We measure effectiveness in terms of correctly solved tasks and efficiency in terms of CPU time, which are standard metrics in the software verification community and used by SV-COMP and Test-Comp. For witness validation, we use the two active SV-COMP validators that support the required witness features, ensuring that our results reflect the actual acceptance criteria of the competition.

5 Conclusion

We presented Test-to-Witness, a tool to convert test cases into violation witnesses. Our extensive evaluation shows that the transformation of test suites is feasible and efficient. We provided evidence that even the combination of a simple plain random tester with any state-of-the-art verifier can improve the effectiveness of bug-finding capabilities in formal verification significantly. Our experiments suggest that especially tools that focus heavily on proving programs correct benefit from a pairing with a test generator. We also show that a portfolio approach, with a small negative impact on efficiency, can improve effectiveness of all considered verifiers.

Data Availability. Test-to-Witness and AFL-TC are publicly available via our supplementary web page https://www.sosy-lab.org/research/test-to-witness/ and as reproduction package at Zenodo [59]. The reproduction package was evaluated by the artifact-evaluation committee and allows the reproduction of our experiments and contains all reported data.

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

References

1. 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
2. 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
3. Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proc. ATVA. pp. 189–208. LNCS 11781, Springer (2019). https://doi.org/10.1007/978-3-030-31784-3_11
4. Alshmrany, K.M., Aldughaim, M., Bhayat, A., Cordeiro, L.C.: FuSeBMC: An energy-efficient test generator for finding security vulnerabilities in C programs. In: Proc. TAP. pp. 85–105. Springer (2021). https://doi.org/10.1007/978-3-030-79379-1_6
5. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. NDSS. The Internet Society (2008), https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/
6. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proc. PLDI. pp. 213–223. ACM (2005). https://doi.org/10.1145/1065010.1065036
7. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: Proc. TAP. pp. 134–153. LNCS 4966, Springer (2008). https://doi.org/10.1007/978-3-540-79124-9_10
8. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. NDSS. Internet Society (2016). https://doi.org/10.14722/ndss.2016.23368
9. Chen, C., Kande, R., Nguyen, N., Andersen, F., Tyagi, A., Sadeghi, A., Rajendran, J.: Hypfuzz: Formal-assisted processor fuzzing. In: USENIX Security. pp. 1361–1378. USENIX Association (2023)
10. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013). https://doi.org/10.1145/2408776.2408795
11. Metta, R., Medicherla, R.K., Chakraborty, S.: BMC+Fuzz: Efficient and effective test generation. In: Proc. DATE. pp. 1419–1424. IEEE (2022). https://doi.org/10.23919/DATE54114.2022.9774672
12. Beyer, D., Lemberger, T.: Six years later: Testing vs. model checking. Int. J. Softw. Tools Technol. Transf. **26**(6), 633–646 (2024). https://doi.org/10.1007/S10009-024-00769-8
13. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15

14. Beyer, D.: Automatic testing of C programs: Test-Comp 2024. Springer (2024)
15. Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction and test generation. In: Proc. ASE. pp. 1138–1141. IEEE (2019). https://doi.org/10.1109/ASE.2019.00121
16. Collection of verification tasks. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks, accessed: 2025-10-17
17. Beyer, D., Strejček, J.: Improvements in software verification and witness validation: SV-COMP 2025. In: Proc. TACAS (3). pp. 151–186. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_9
18. Beyer, D.: Advances in automatic software testing: Test-Comp 2025. In: Proc. FASE. pp. 257–274. LNCS 15693, Springer (2025). https://doi.org/10.1007/978-3-031-90900-9_13
19. Beyer, D.: Second competition on software testing: Test-Comp 2020. In: Proc. FASE. pp. 505–519. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_25
20. Beyer, D.: Status report on software testing: Test-Comp 2021. In: Proc. FASE. pp. 341–357. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_17
21. Beyer, D.: Advances in automatic software testing: Test-Comp 2022. In: Proc. FASE. pp. 321–335. LNCS 13241, Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_18
22. Beyer, D.: Software testing: 5th comparative evaluation: Test-Comp 2023. In: Proc. FASE. pp. 309–323. LNCS 13991, Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_17
23. Google: american fuzzy lop. https://github.com/google/AFL, accessed: 2025-10-17
24. Böhme, M., Pham, V., Roychoudhury, A.: Coverage-based greybox fuzzing as markov chain. In: Proc. SIGSAC. pp. 1032–1043. ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978428
25. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC. pp. 99–114. LNCS 10629, Springer (2017). https://doi.org/10.1007/978-3-319-70389-3_7
26. Yoo, S., Harman, M.: Regression testing minimization, selection, and prioritization: A survey. STVR **22**(2), 67–120 (2012). https://doi.org/10.1002/stvr.430
27. McMinn, P.: Search-based software test-data generation: A survey. STVR **14**(2), 105–156 (2004). https://doi.org/10.1002/stvr.294
28. Li, J., Zhao, B., Zhang, C.: Fuzzing: A survey. Cybersecurity **1**(1), 6 (June 2018). https://doi.org/10.1186/s42400-018-0002-y
29. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Trans. Software Eng. **47**(11), 2312–2331 (2021). https://doi.org/10.1109/TSE.2019.2946563
30. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic-execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018). https://doi.org/10.1145/3182657
31. Anand, S., Burke, E.K., Chen, T.Y., Clark, J.A., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software **86**(8), 1978–2001 (2013). https://doi.org/10.1016/j.jss.2013.02.061
32. D'Silva, V., Kröning, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. on CAD of Integrated Circuits and Systems **27**(7), 1165–1178 (2008). https://doi.org/10.1109/TCAD.2008.923410

33. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys **41**(4) (2009). https://doi.org/10.1145/1592434.1592438
34. Garavel, H., ter Beek, M.H., van de Pol, J.: The 2020 expert survey on formal methods. In: Proc. FMICS. pp. 3–69. LNCS 12327, Springer (2020). https://doi.org/10.1007/978-3-030-58298-2_1
35. Beyer, D., Podelski, A.: Software model checking: 20 years and beyond. In: Principles of Systems Design. pp. 554–582. LNCS 13660, Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_27
36. Godefroid, P., Sen, K.: Combining model checking and testing. In: Handbook of Model Checking, pp. 613–649. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_19
37. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook of Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
38. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: A survey. STVR **19**(3), 215–261 (2009). https://doi.org/10.1002/stvr.402
39. Kaleeswaran, A.P., Nordmann, A., Vogel, T., Grunske, L.: A systematic literature review on counterexample explanation. Information and Software Technology **145**, 106800 (2022). https://doi.org/10.1016/j.infsof.2021.106800
40. Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Proc. CAV (2). pp. 502–509. LNCS 9780, Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_28
41. Novikov, E., Zakharov, I.S.: Towards automated static verification of GNU C programs. In: Proc. PSI. pp. 402–416. LNCS 10742, Springer (2017). https://doi.org/10.1007/978-3-319-74313-4_30
42. Groce, A., Kröning, D., Lerda, F.: Understanding counterexamples with explain. In: Proc. CAV'04. pp. 453–456. LNCS 3114, Springer (2004). https://doi.org/10.1007/978-3-540-27813-9_35
43. 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
44. Chaki, S., Groce, A., Strichman, O.: Explaining abstract counterexamples. In: Proc. FSE. pp. 73–82. ACM (2004). https://doi.org/10.1145/1029894.1029908
45. Castaño, R., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Model checker execution reports. In: Proc. ASE. pp. 200–205. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115633
46. 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
47. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
48. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. pp. 184–203. LNCS 14624, Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_11
49. Beyer, D., Strejček, J.: SV-Witnesses – Format 2.1. Zenodo (2025). https://doi.org/10.5281/zenodo.17277275
50. Beyer, D., Jakobs, M.C.: Cooperative verifier-based testing with CoVeriTest. Int. J. Softw. Tools Technol. Transfer **23**(3), 313–333 (2021). https://doi.org/10.1007/s10009-020-00587-8

51. Barth, M., Jakobs, M.C.: Test-case generation with automata-based software model checking. In: Proc. SPIN. Springer (2024). https://doi.org/10.1007/978-3-031-66149-5_14
52. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008), available at https://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
53. Lemberger, T.: Plain random test generation with PRTest (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 871–873 (December 2021). https://doi.org/10.1007/s10009-020-00568-x
54. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
55. Beyer, D., Strejček, J.: SV-Benchmarks: Benchmark set for software verification (SV-COMP 2025). Zenodo (2025). https://doi.org/10.5281/zenodo.15012096
56. 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
57. Beyer, D., Lemberger, T.: TestCov: Robust test-suite execution and coverage measurement. In: Proc. ASE. pp. 1074–1077. IEEE (2019). https://doi.org/10.1109/ASE.2019.00105
58. Wilcoxon, F.: Individual comparisons by ranking methods. Biometrics Bulletin **1**(6), 80–83 (1945). https://doi.org/10.2307/3001968
59. Beyer, D., Lemberger, T., Wachowitz, H.: Artifact for the fase 26 paper: Testing in formal verification via witness generation (empirical evaluation). Zenodo (2026). https://doi.org/10.5281/zenodo.18351121