Evaluating Tools for Automatic Software Testing (Report on Test-Comp 2026)

Dirk Beyer

LMU Munich, Munich, Germany

Abstract. This report presents the results of the 8th Competition on Software Testing (Test-Comp 2026), which is an annual event to provide an overview and comparative evaluation of automatic tools for test-suite generation for C programs. The experiment setup is publicly available and suitable for reuse as a baseline when comparing newly developed approaches for test generation. The benchmark set SV-Benchmarks contains 16 217 test-generation tasks for C programs. Each test-generation task consists of a program and a test specification. The test specifications included error coverage (generate a test suite that exhibits a bug) and branch coverage (generate a test suite that executes as many program branches as possible). Test-Comp 2026 evaluated 21 software systems for test generation that are all freely available. This included 11 test-suite generators that participated with active support from teams led by 11 different representatives from 6 countries (actively maintained software systems, participation in competition jury). Test-Comp 2026 had 1 new test generator (AFL-to-TC[new]) and 1 new test-suite validator (TestCoCa[new]). The evaluation included also 10 test-generation tools from previous years.

Keywords: Software Testing · Test-Case Generation · Competition · Program Analysis · Software Validation · Software Bugs · Test Validation · Test-Comp · Benchmarking · Test Coverage · Bug Finding · Test Suites · SV-Benchmarks · BenchExec · TestCov · FM-Weck

1 Introduction

This report explains the competition setup and presents the results of the 8th edition of the International Competition on Software Testing (Test-Comp, `https://test-comp.sosy-lab.org`). The competition compares automatic test-suite generators for C programs, in order to showcase the state of the art in the area of automatic software testing. Since this report is a continuation of the series of yearly competition reports, we only slightly adjust the structure of the new report,

This report extends previous reports on Test-Comp [11, 13, 14, 15, 16, 18, 19, 21] by providing new results, while the procedures and setup of the competition stay mainly unchanged. Reproduction packages are available on Zenodo (see Table 5).

✉ `dirk.beyer@sosy.ifi.lmu.de`

in order to facilitate quick lookup as reference [11, 13, 14, 15, 16, 18, 19, 21]. We repeat the rules and definitions, present the competition results, and give some interesting data about the execution of the competition experiments.

Competition Goals. In summary, the goals of Test-Comp are the following [13]:

- Establish *standards* for software test generation. This means, most prominently, to develop a standard for marking input values in programs, define an exchange format for test suites, agree on a specification language for test-coverage criteria, and define how to validate the resulting test suites.
- Establish a *benchmark* set for software testing in the community. This means to create and maintain a set of programs together with coverage criteria, and to make those publicly available for researchers to be used in performance comparisons when evaluating a new technique.
- Provide an overview of *available tools* for test-case generation and a snapshot of the state-of-the-art in software testing to the community. This means to compare, independently from particular paper projects and specific techniques, different test generators in terms of effectiveness and performance.
- Increase the visibility and credits that *tool developers* receive. This means to provide a forum for presentation of tools and discussion of the latest technologies, and to give the participants the opportunity to publish about the development work that they have done.
- Educate PhD students and other participants on how to set up performance experiments, package tools in a way that supports reproduction, and how to perform *robust and accurate research experiments*.
- Provide *resources* to development teams that do not have sufficient computing resources and give them the opportunity to obtain results from experiments on large benchmark sets.

Related Competitions. In the field of formal methods, competitions are respected as an important evaluation method and there are many competitions [9, 31]. We refer to the report from Test-Comp 2020 [13] for a more detailed discussion and give here only the references to the most related competitions: Competition on Software Verification (SV-COMP) [36], Competition on Search-Based Software Testing (SBST) [62], and the DARPA Cyber Grand Challenge [65]. An overview of competitions can be found in the TOOLympics volumes [9, 31]. For the techniques used for automatic software testing, we refer to the literature [5, 47].

Quick Summary of Changes. While we keep the setup of the competition stable, we apply some consolidation and extensions to improve the quality of the competition. The following changes were made for Test-Comp 2026:

- The registration to participate and the qualification process were simplified.
- We use a more systematic way to name the categories, consisting of the language, the coverage specification, and the name of the base category.
- The number of test-generation tasks was increased from 11 226 to 16 217.
- Several new base categories were added.

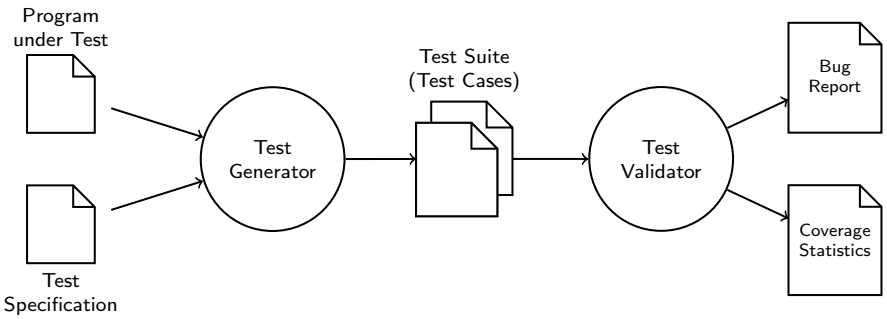

Fig. 1: Flow of the Test-Comp execution for one test generator (taken from [13])

2 Organization, Definitions, Formats, and Rules

Organizational aspects such as the classification (automatic, off-site, reproducible, jury, training) and the competition schedule is given in the initial competition definition [11]. In the following, we repeat some important definitions that are necessary to understand the results.

Organizer and Jury. The competition Test-Comp consists of an organizer, the participants, and a jury. The organizer is responsible for hosting a web site, executing the experiments, providing the results for the participants, and assembling the competition report after the competition is completed. The competition jury is responsible for overseeing the process, ensuring transparency of all components of the competition, resolving any interpretation questions regarding the rules, and reviewing the competition-contribution papers, from which a selection will be published in the FASE proceedings.

Test-Generation Task. A *test-generation task* is a pair of an input program (program under test) and a test specification. A *test-generation run* is a non-interactive execution of a test generator on a single test-generation task, in order to generate a test suite according to the test specification. A *test suite* is a sequence of test cases, given as a directory of files according to the format for exchangeable test-suites.[1]

Execution of a Test Generator. Figure 1 illustrates the process of executing one test-suite generator on one benchmark test-generation task. One test run for a test-suite generator gets as input (i) a program from the benchmark set and (ii) a test specification (cover error, or cover branches), and returns as output a test suite (i.e., a set of test cases). The test generator is contributed by a competition participant as a software archive in ZIP format on Zenodo, via a DOI entry of a version in the FM-Tools record of the test generator. All test runs are executed centrally by the competition organizer.

[1] https://gitlab.com/sosy-lab/test-comp/test-format

Table 1: Coverage specifications used in Test-Comp 2026 (same as 2019–2025)

Formula	Interpretation
`COVER EDGES(@CALL(reach_error))`	The test suite contains at least one test that executes function `reach_error`.
`COVER EDGES(@DECISIONEDGE)`	The test suite contains tests such that all branches of the program are executed.

Execution of the Test Validator. The test-suite validator takes as input the test suite from the test generator and validates it by executing the program on all test cases: for bug finding it checks if the bug is exposed and for coverage it reports the coverage. In Test-Comp 2026, we used the test-suite validators TestCov [34] (in four configurations as in Test-Comp 2025 [21]) and TestCoCa[new] (in one configuration).

Test Specification. The specification for testing a program is given to the test generator as input file (either `properties/coverage-error-call.prp` or `properties/coverage-branches.prp` for Test-Comp 2026, as previously). The definition `init(main())` is used to define the initial states of the program under test by a call of function `main` (with no parameters). The definition `FQL(f)` specifies that coverage definition `f` should be achieved. The FQL (FShell query language [51]) coverage definition `COVER EDGES(@DECISIONEDGE)` means that all branches should be covered (typically used to obtain a standard test suite for quality assurance) and `COVER EDGES(@CALL(foo))` means that a call (at least one) to function `foo` should be covered (typically used for bug finding). A complete specification looks like: `COVER(init(main()), FQL(COVER EDGES(@DECISIONEDGE)))`.

Table 1 lists the two FQL formulas that are used in test specifications of Test-Comp 2026; there was no change from 2020 (except that special function `__VERIFIER_error` does not exist anymore).

Task-Definition Format 2.1. Test-Comp 2026 used the task-definition format in version 2.1.

License and Qualification. The license of each participating test generator must allow its free use for reproduction of the competition results. The license for each tool is available in the FM-Tools entry for the tool, as well as in Table 2. Details on qualification criteria can be found in the competition report of Test-Comp 2019 [14].

Technical Setup and Infrastructure. The technical setup of the competition is based on BenchExec [35] to execute the benchmark runs, BenchCloud [29] to distribute the execution to a large and elastic set of computers, FM-Weck [37] to execute tools (even from from previous years) using a container with all their requirements fulfilled, and the FM-Tools [22] collection to look up all the information we need about the tools for test-case generation, including their versions, parameters, and jury representatives. The results are presented in tables and graphs, also on the competition web site (`https://test-comp.sosy-lab.org/2026/results`), and are available in the accompanying archives (see Table 5).

Table 2: Evaluated tools (test generators and test-suite validators) with tool references and representing jury members; new indicates first-time participants, ∅ indicates inactive (hors concours) participation; licenses are abbreviated, see the hyperlink or tool page at FM-Tools for the specific version of the license; TestCoCa new and TestCov are the test-suite validators that compute the scores for each test-suite

Tester	Ref.	License	Jury member	Affiliation
AFL-to-TC new	[58]	Apache	H. Wachowitz	LMU Munich, Germany
cetfuzz ∅		Apache	–	–
CoVeriTest	[32, 53]	Apache	M.-C. Jakobs	LMU Munich, Germany
ESBMC-incr	[69]	Apache	C. Wei	U. Manchester, UK
ESBMC-kind	[48, 69]	Apache	C. Wei	U. Manchester, UK
FDSE	[71, 72]	Apache	Z. Chen	National U. Defense Techn., China
Fizzer	[54, 55]	Zlib	M. Trtík	Masaryk U., Brno, Czechia
FuSeBMC	[3, 4]	MIT	K. Alshmrany	Inst. Public Admin., Saudi Arabia
FuSeBMC-AI ∅	[1, 2]	MIT	–	–
HybridTiger ∅	[39, 64]	Apache	–	–
KLEE ∅	[40, 41]	NCSA	–	–
KLEEF ∅	[61]	NCSA	–	–
Owi ∅		AGPL	–	–
PRTest	[33, 57]	Apache	T. Lemberger	LMU Munich, Germany
Rizzer ∅		Zlib	–	–
Sikraken	[60]	LGPL	C. Meudec	South East Technological U., Ireland
Symbiotic	[42, 43]	MIT	M. Jonáš	Masaryk U., Brno, Czechia
TracerX ∅	[45, 52]	Apache	–	–
TracerX-WP ∅	[45, 52]	Apache	–	–
UTestGen	[7, 8]	LGPL	M. Barth	LMU Munich, Germany
WASP-C ∅	[59]	Apache	–	–
TestCoCa new	[46]	Zlib	M. Trtík	Masaryk U., Brno, Czechia
TestCov	[34]	Apache	M. Kettl	LMU Munich, Germany

Participating Test-Suite Generators and Test-Suite Validators. We provide an overview of the participating test generators and test-suite validators in Table 2. The table lists the tool name together with a hyperlink to the FM-Tools page for the tool, references to their publications, the license of the tool, the team representatives of the jury of Test-Comp 2026, and their affiliation. (The competition jury consists of the chair and one member of each participating team.) An online table with information about all participating systems is provided on the competition web site.[2] Table 3 lists the features and technologies that are used in the test generators, as declared in the FM-Tools record for each tool.

There are test generators that did not actively participate (tester archives taken from last year) and that are not included in rankings. Those are called *inactive* participation and the tools are labeled with a symbol (∅). In the past, we named those inactive tools 'hors concours', but since there could be other

[2] https://test-comp.sosy-lab.org/2026/systems.php

Table 3: Algorithms and techniques used by the participating tools; we use the annotations ⌀ for inactive and new for first-time participants

Tool	Bit-Precise Analysis	Bounded Model Checking	CEGAR	Concurrency Support	Explicit-Value Analysis	Floating-Point Arithmetics	Guidance by Coverage Measures	Interpolation	k-Induction	Portfolio	Predicate Abstraction	Random Execution	Symbolic Execution	Targeted Input Generation
CoVeriTest	✓		✓		✓	✓				✓	✓	✓		
ESBMC-incr	✓	✓		✓					✓					
ESBMC-kind	✓	✓		✓	✓	✓			✓					
FDSE						✓	✓					✓	✓	
Fizzer	✓													
FuSeBMC		✓				✓	✓			✓				✓
FuSeBMC-AI⌀		✓				✓	✓			✓				✓
HybridTiger⌀			✓		✓	✓					✓			
KLEE⌀						✓							✓	✓
KLEEF⌀	✓					✓	✓						✓	✓
Owi⌀	✓					✓						✓	✓	✓
PRTest						✓						✓		
Rizzer⌀	✓												✓	
Sikraken													✓	
Symbiotic	✓			✓		✓	✓	✓	✓	✓	✓		✓	✓
TracerX⌀		✓				✓		✓					✓	✓
TracerX-WP⌀								✓						
UTestGen			✓					✓		✓				
WASP-C⌀						✓						✓	✓	

reasons for hors-concours participation (for example meta tools that consist of other participating tools), we now use the more specific term 'inactive'.

3 Categories and Scoring Schema

Benchmark Programs. The input programs were taken from the largest and most diverse open-source repository of software-verification and test-generation tasks [3], which is also used by SV-COMP [20]. As in the previous editions, we selected all programs for which the following properties were satisfied (see merge

[3] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

requests on GitLab for Test-Comp 2019 [4], and for Test-Comp 2026 [5], and the Test-Comp 2019 report [14]):

1. compiles with `gcc`, if a harness for the special methods [6] is provided,
2. should contain at least one call to a nondeterministic function,
3. does not rely on nondeterministic pointers,
4. does not have expected result 'false' for property 'termination', and
5. has expected result 'false' for property 'unreach-call' (only for category *Cover-Error*).

This selection yielded a total of 16 217 test-generation tasks, namely 1 895 tasks for category *Cover-Error* and 14 322 tasks for category *Cover-Branches*.

Categories. The test-generation tasks are partitioned into categories, which are listed in Tables 6 and 7 and described in detail on the competition web site.[7] Figure 2 illustrates the category composition.

Changes to Category Structure in Test-Comp 2026. In the following we outline the most important changes to the categories used for the competition:

- We use a more systematic way to name the categories, consisting of the language (always C for Test-Comp), the coverage specification (coverage-error-call or coverage-branches): ⟨*language*⟩.⟨*specification*⟩.⟨*base-category*⟩. For example, the complete name of a category looks like: *C.coverage-branches.BitVectors*.
- The previous base category *Fuzzle* was merged into base category *Recursive* in order to better balance the category structure. [8] Therefore, *Fuzzle* was removed from both *C.Cover-Error* and *C.Cover-Branches*.
- Base category *SoftwareSystems-coreutils* was added to *C.Cover-Error*.
- Base category *SoftwareSystems-Intel-TDX-Module* was added to both *C.Cover-Error* and *C.Cover-Branches*. This alone added 658 tasks to *C.Cover-Error* and 836 tasks to *C.Cover-Branches*. This category is special because it consists of security-critical firmware code from the Intel® TDX Module. More details about these verification and test tasks are available from a case-study [27, 28]. These tasks use the newly introduced competition-specific function `__VERIFIER_nondet_memory(void *, size_t)`, which havocs a given memory block of a given size.

Category Cover-Error. The first category is to show the abilities to discover bugs. The benchmark set consists of programs that contain a bug. We produce for every tool and every test-generation task one of the following scores: 1 point if the validator succeeds in executing the program under test on a generated test case that explores the bug (i.e., the specified function was called), and 0 points otherwise.

[4] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/774
[5] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1683
[6] https://test-comp.sosy-lab.org/2026/rules.php
[7] https://test-comp.sosy-lab.org/2026/benchmarks.php
[8] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1599

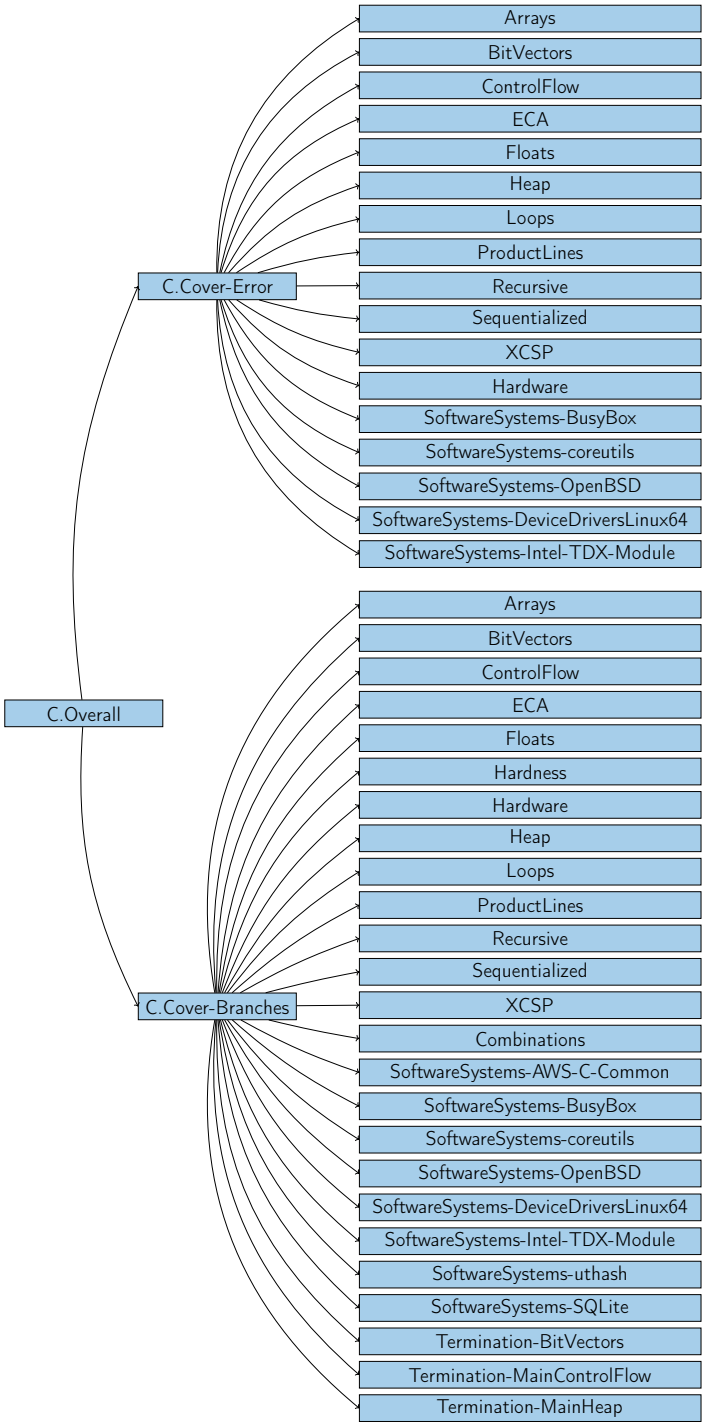

Fig. 2: Category structure for Test-Comp 2026

Category Cover-Branches. The second category is to cover as many branches of the program as possible. The coverage criterion was chosen because many test generators support this standard criterion by default. Other coverage criteria can be reduced to branch coverage by transformation [50]. We produce for every tool and every test-generation task the coverage of branches of the program (as reported by TestCov [34] and TestCoCa[new]; a value between 0 and 1) that are executed for the generated test cases. The score is the returned coverage.

Max Over All Validators. As mentioned before, each test-suite is validated five times: TestCov is executed four times on each test suite, using four different configurations, and TestCoCa[new] is executed once. The score of a test suite is the maximum of the four computed scores. This reduces the validation bias towards a particular technique: the five configurations together can validate more coverage then a single technique.

Ranking. The ranking was decided based on the sum of points (normalized for meta categories). In case of a tie, the ranking was decided based on the run time, which is the total CPU time over all test-generation tasks. Opt-out from categories was possible and scores for categories were normalized based on the number of tasks per category (see competition report of SV-COMP 2013 [10], page 597).

4 Reproducibility

We followed the same competition workflow that was described in detail in a previous competition report (see [15, Sect. 4]). All major components that were used for the competition were made available in public version-control repositories. An overview of the components that contribute to the reproducible setup of Test-Comp is provided in Fig. 3, and the details are given in Table 4. We refer to the report of Test-Comp 2019 [14] for a thorough description of all components of the Test-Comp organization and how we ensure that all parts are publicly available for maximal reproducibility. In order to guarantee long-term availability and immutability of the test-generation tasks, the produced competition results, and the produced test suites, we also packaged the material and published it at Zenodo (see Table 5). The competition report of SV-COMP 2022 (see [17, Sect. 3]) provides a description on reproducing individual results and on trouble-shooting.

OCI Containers. The competition used FM-Weck [37][9] again to provide participants easy execution of their tools inside the competition container. Test-Comp 2026 executed all tools (active and inactive, test-generation and test-validation tools) in containers using FM-Weck [37] (which is based on podman). In Test-Comp 2025, the container solution was used only for the inactive tools. This was necessary to include older archives in the comparative evaluation, even if the tools were made for an older distribution of Ubuntu or use packages that are not available anymore. Now, also active tools are executed in containers, and the tools can specify in their FM-Tools [22] entry a container in which they can run.

[9] https://gitlab.com/sosy-lab/software/fm-weck

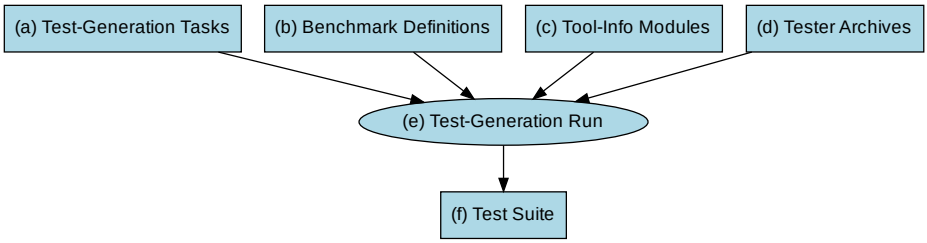

Fig. 3: Benchmarking components of Test-Comp and competition's execution flow (same as for Test-Comp 2020)

Table 4: Publicly available components for reproducing Test-Comp 2026

Component	Fig. 3	Repository at `http://gitlab.com/sosy-lab/...`	Version
Test-Generation Tasks	(a)	`benchmarking/sv-benchmarks`	`testcomp26`
Benchmark Definitions	(b)	`test-comp/bench-defs`	`testcomp26`
Tool-Info Modules	(c)	`software/benchexec`	`3.34`
Test-Generators	(d)	`benchmarking/fm-tools`	`2.3`
BenchExec (Benchmarking)	(e)	`software/benchexec`	`3.34`
BenchCloud (Distribution)	(e)	`gitlab.com/sosy-lab/software/benchcloud`	`1.5.0`
FM-Weck (Containers)	(e)	`gitlab.com/sosy-lab/software/fm-weck`	`1.6.0`
Test-Suite Format	(f)	`test-comp/test-format`	`testcomp26`
Processing Scripts		`benchmarking/competition-scripts`	`testcomp26`

Table 5: Artifacts published for Test-Comp 2026

Content	DOI	Reference
Test-Generation Tasks	`10.5281/zenodo.18650775`	[25]
Competition Results	`10.5281/zenodo.18650772`	[24]
Test-Suite Generators	`10.5281/zenodo.18650756`	[23]
Test Suites (Witnesses)	`10.5281/zenodo.18650733`	[26]
BenchExec	`10.5281/zenodo.18455156`	[70]
FM-Weck	`10.5281/zenodo.18650812`	[38]
BenchCloud	`10.5281/zenodo.14331949`	[6]

For example, consider the FM-Tools snippet in Fig. 4: Besides the version id, the DOI of the archive, and the command-line parameters, the key `full_container_images` specifies the container to be used to execute the tool. The OCI container image is long-term archived under `doi:10.5281/zenodo.18001984` at Zenodo.

Files at Zenodo not Immutable Anymore. The immutability of a uniquely identified object is of existential importance for reproducibility. The reproducibility of the experiments of the competition relies on the property that the content of a tool archive (of a test-generation tool) does not change after its registration.

```yaml
versions:
  - version: "testcomp26"
    doi: "10.5281/zenodo.18051027"
    benchexec_toolinfo_options: ["-s", "incr"]
    required_ubuntu_packages: []
    base_container_images:
      - docker.io/ubuntu:24.04
    full_container_images:
      - registry.gitlab.com/sosy-lab/benchmarking/fm-tools/10.5281/zenodo.18001984
```

Fig. 4: Snippet of FM-Tools data file for FuSeBMC for the tool version (from file `fusebmc.yml`, version 2.3)

We assumed that the content for a given version-specific DOI is immutable, which was the case for Zenodo records for years.

This assumption does unfortunately not hold anymore for Zenodo DOIs since a recent upgrade of the platform and the policies [63]. Zenodo allows users to change the files that are stored for a given DOI: "You can edit the files of your records within 30 days of publishing. Once unlocked you will have 40 days to publish your changes." Even more dramatic, Zenodo now allows users to delete published content: "You can delete your records within 30 days of publishing." [Zenodo, 2026-02-21]

To fix this threat to reproducibility, we cannot use DOIs for specific versions as identifiers anymore. We need to use *content identifiers* such as cryptographic hashes (e.g., SHA-256) or hash trees (e.g., Merkle trees) of the archive files in order to uniquely identify them. The API of Zenodo provides unfortunately only MD5 hashes for the files that it stores.

Currently, the competition scripts store the version hashes for the relevant repositories of the competition environment (fm-tools, sv-benchmarks, benchexec, competition-scripts, bench-defs), the archive (name, DOI suffix, date, hash), and the container, inside the tag `<description>` under `<result>` in each results XML file that BenchExec produces. (For example, have a look at such a results file from the results artifact [12].) This description tagging was meant for humans, e.g., reproduction engineers, who want to double check the correct components. So far, it was assumed that the identifier from the DOI is sufficient to address an immutable archive, but now we should add new XML tags to store a hash or hash tree. This way, it can be (automatically) checked that an archive file is indeed the same archive file that was used for the competition experiments.

Alternatively, one could use Zenodo as long-term storage for the archive files, but address and access the files via IPFS. That is, tool archives get published at Zenodo as before, and tool archives are then specified in the FM-Tools record by the DOI *and* a content identifier.[10]

[10] Another thought was to use the Software Heritage Archive, but this archive is restricted to source code and to public version-control repositories only.

5 Results and Discussion

This section represents the results of the competition experiments. The report shall help to understand the state of the art and the advances in fully automatic test generation for whole C programs, in terms of effectiveness (test coverage, as accumulated in the score) and efficiency (resource consumption in terms of CPU time). All results mentioned in this article were inspected and approved by the participants.

Computing Resources. The computing environment and the resource limits were the same as previously [21]. Each test run was limited to 4 processing units (cores), 15 GB of memory, and 15 min of CPU time. The test-suite validation was limited to 2 processing units, 7 GB of memory, and 5 min of CPU time. The machines for running the experiments are part of a compute cluster that consists of 168 machines. Each machine had one Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 24.04 with Linux kernel 6.8). We used BenchExec [35] to measure and control computing resources (CPU time, reponse time), BenchCloud [29] to distribute, install, run, and clean-up test-case generation runs, and to collect the results, and FM-Weck [37] to execute the tool in the correct container according to the tool's FM-Tools [22] entry. The values for CPU time are accumulated over all cores of the CPU. Further technical parameters of the competition machines are available in the repository that contains the competition scripts. [11] The packages that were installed in the competition execution container are also specified there. [12]

One complete test-generation execution of the competition consisted of 325 560 single test-generation run executions. The total CPU time was 5.3 years for one complete competition run for test generation (without validation). Test-suite validation consisted of 1 627 523 single test-suite validation runs. The total consumed CPU time was 1.6 years. Each tool was executed several times, for training and in order to make sure no installation issues occur during the execution. Including preruns, the infrastructure managed a total of 1.1 million single test-generation runs (consuming 18 years of CPU time). The prerun test-suite validation consisted of 3.9 million single test-suite validation runs (consuming 3.1 years of CPU time).

Quantitative Results. The quantitative results are presented in the same way as last year: Table 6 presents the quantitative overview of all tools and all categories. The head row mentions the category and the number of test-generation tasks in that category. The tools are listed in alphabetical order; every table row lists the scores of one test generator. We indicate the top three candidates by formatting their scores in bold face and in larger font size. An empty table cell means that the test generator opted-out from the respective main category (perhaps participating in subcategories only, restricting the evaluation to a specific topic). More information (including interactive tables, quantile plots for every category, and also the raw data in XML format) is available on

[11] https://gitlab.com/sosy-lab/benchmarking/competition-scripts/-/tree/svcomp26?ref_type=tags#parameters-of-runexec

[12] .../benchmarking/competition-scripts/-/blob/svcomp26/test/Dockerfile.user.2026

Table 6: Quantitative overview over all results; empty cells mark opt-outs; new indicates first-time participants, ∅ indicates hors-concours participation

Participant	C.Cover-Error 1895 tasks max. score 1895	C.Cover-Branches 14322 tasks max. score 14322	C.Overall 16217 tasks max. score 16217
AFL-to-TC new	1082	6817	8488
cetfuzz ∅	556	3584	4410
CoVeriTest	853	7084	7660
ESBMC-incr	445	1378	2685
ESBMC-kind	472	2508	3438
FDSE	1103	**8121**	**9319**
Fizzer	**1123**	**7967**	**9317**
FuSeBMC	**1486**	**8237**	**11024**
FuSeBMC-AI ∅	1338	5960	9099
HybridTiger ∅	727	5688	6329
KLEE ∅	1213	4476	7723
KLEEF ∅	1413	8282	10735
Owi ∅	383	3411	3569
PRTest	331	4443	3932
Rizzer ∅	997		
Sikraken		4460	
Symbiotic	**1151**	6192	8429
TracerX ∅	664	4864	5594
TracerX-WP ∅	543	4679	4974
UTestGen	629	5643	5888
WASP-C ∅	818	3844	5674

the competition web site[13] and in the results artifact (see Table 5). Table 7 reports the top three test generators for each category. The consumed run time (column 'CPU Time') is given in hours.

Score-Based Quantile Functions for Quality Assessment. We use score-based quantile functions [35] because these visualizations make it easier to understand the results of the comparative evaluation. The web site[13] and the results artifact (Table 5) include such a plot for each category; as example, we show the plot for category *C.Overall* (all test-generation tasks) in Fig. 5. We had

[13] https://test-comp.sosy-lab.org/2026/results

Table 7: Overview of the top-three test generators for each category (measurement values for CPU time rounded to two significant digits, in hours)

Rank	Tester	Score	CPU Time
C.Cover-Error (1895 tasks, max. score 1895)			
1	**FuSeBMC**	**1486**	290
2	Symbiotic	1151	160
3	Fizzer	1123	230
C.Cover-Branches (14322 tasks, max. score 14322)			
1	**FuSeBMC**	**8237**	3 500
2	FDSE	8121	3 400
3	Fizzer	7967	2 500
C.Overall (16217 tasks, max. score 16217)			
1	**FuSeBMC**	**11024**	3 800
2	FDSE	9319	3 800
3	Fizzer	9317	2 800

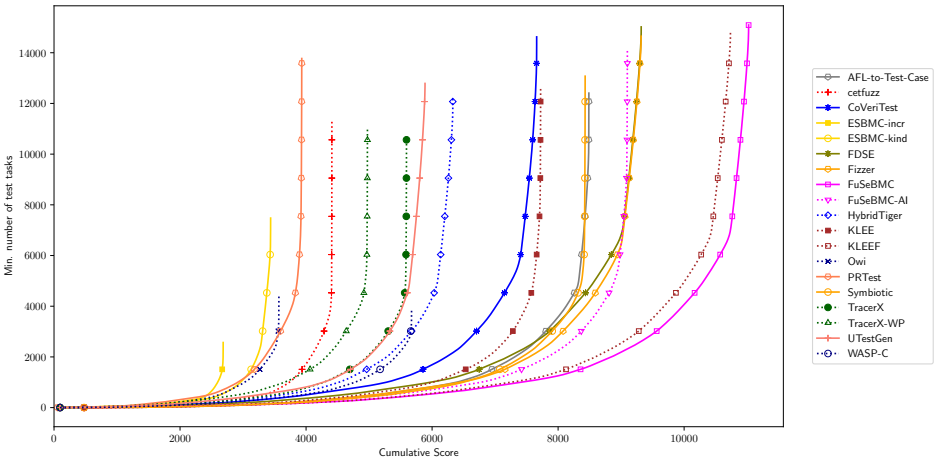

Fig. 5: Quantile functions for category *C.Overall*; each quantile function illustrates the quantile (x-coordinate) of the scores obtained by test-generation runs below a certain number of test-generation tasks (y-coordinate); more details about score-based quantile plots can be looked up in a previous report [14]; the graphs are decorated with symbols to make them better distinguishable without color

18 test generators participating in category *C.Overall*, for which the quantile plot shows the overall performance over all categories (scores for meta categories are normalized [10]). A more detailed discussion of score-based quantile plots for testing is provided in the Test-Comp 2019 competition report [14].

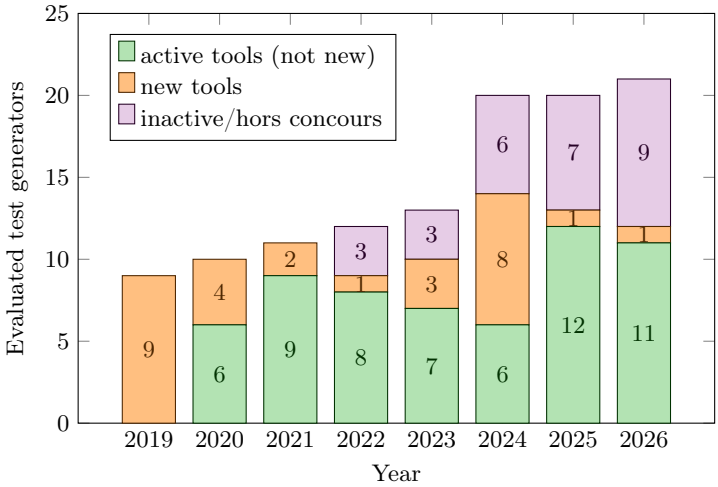

Fig. 6: Number of evaluated test generators for each year (green/bottom: active participants from previous years, orange/middle: number of first-time participants, violet/top: inactive participants from previous years)

6 Conclusion

The 8th Competition on Software Testing (Test-Comp) provides again an overview of fully-automatic test-generation tools for C programs. In total, 21 test-suite generators were compared (see Fig. 6 for the participation numbers and Table 2 for the details). This off-site competition uses a benchmark infrastructure that makes the execution of the experiments fully-automatic and reproducible. Transparency is ensured by making all components available in public repositories and by having a jury (consisting of members from each team) that oversees the process. All test suites were validated by the test-suite validator TestCov [34] and by the newly contributed test-suite validator TestCoCa[new] to measure the coverage. The competition used again several different validation runs for each test suite, in order to obtain the best possible coverage result, using different compiler backends and different formatting choices after instrumentation for coverage measurement. The results of the competition were presented at the 29th International Conference on Fundamental Approaches to Software Engineering (FASE) at ETAPS 2026 in Turin, Italy.

Data-Availability Statement. The test-generation tasks and results of the competition are published at Zenodo, as described in Table 5. All components and data that are necessary for reproducing the competition are available in public version repositories, as specified in Table 4. For easy access, the results are presented also online on the competition web site `https://test-comp.sosy-lab.org/2026/results`.

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 418257054 (Coop).

References

1. Aldughaim, M., Alshmrany, K.M., Gadelha, M.R., de Freitas, R., Cordeiro, L.C.: FuSeBMC_IA: Interval analysis and methods for test-case generation (competition contribution). In: Proc. FASE. pp. 324–329. LNCS 13991, Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_18
2. Aldughaim, M., Alshmrany, K.M., Mustafa, M., Cordeiro, L.C., Stancu, A.: Bounded model checking of software using interval methods via contractors. arXiv/CoRR **2012**(11245) (December 2020). https://doi.org/10.48550/arXiv.2012.11245
3. Alshmrany, K., Aldughaim, M., Cordeiro, L., Bhayat, A.: FuSeBMC v.4: Smart seed generation for hybrid fuzzing (competition contribution). In: Proc. FASE. pp. 336–340. LNCS 13241, Springer (2022). https://doi.org/10.1007/978-3-030-99429-7_19
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. 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
6. Barth, M., Beyer, D., Chien, P.C., Jankola, M.: Benchcloud Release 1.5.0. Zenodo (2026). https://doi.org/10.5281/zenodo.14331949
7. Barth, M., Dietsch, D., Heizmann, M., Jakobs, M.C.: Ultimate TestGen: Combining parallel trace abstraction and symbolic path execution (competition contribution). In: Proc. FASE. LNCS 16504, Springer (2026)
8. 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
9. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: TOOLympics 2019: An overview of competitions in formal methods. In: Proc. TACAS (3). pp. 3–24. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_1
10. Beyer, D.: Second competition on software verification (Summary of SV-COMP 2013). In: Proc. TACAS. pp. 594–609. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_43
11. Beyer, D.: Competition on software testing (Test-Comp). In: Proc. TACAS (3). pp. 167–175. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_11
12. Beyer, D.: Results of the 1st International Competition on Software Testing (Test-Comp 2019). Zenodo (2020). https://doi.org/10.5281/zenodo.3856661
13. 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
14. Beyer, D.: First international competition on software testing (Test-Comp 2019). Int. J. Softw. Tools Technol. Transf. **23**(6), 833–846 (December 2021). https://doi.org/10.1007/s10009-021-00613-3
15. 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

16. 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
17. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). pp. 375–402. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_20
18. 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
19. Beyer, D.: Automatic testing of C programs: Test-Comp 2024. Springer (2024)
20. 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
21. 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
22. Beyer, D.: Find, use, and conserve tools for formal methods. In: Proc. Festschrift Podelski 65th Birthday. pp. 75–91. LNCS 14765, Springer (2026). https://doi.org/10.1007/978-3-032-13711-1_5
23. Beyer, D.: FM-Tools Release 2.3: Data set of metadata about tools for formal methods (SV-COMP 2026, Test-Comp 2026). Zenodo (2026). https://doi.org/10.5281/zenodo.18650756
24. Beyer, D.: Results of the 8th Intl. Competition on Software Testing (Test-Comp 2026). Zenodo (2026). https://doi.org/10.5281/zenodo.18650772
25. Beyer, D.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2026, Test-Comp 2026). Zenodo (2026). https://doi.org/10.5281/zenodo.18650775
26. Beyer, D.: Test suites from test-generation tools (Test-Comp 2026). Zenodo (2026). https://doi.org/10.5281/zenodo.18650733
27. Beyer, D., Chien, P.C., Huang, B.Y., Lee, N.Z., Lemberger, T.: A case study in firmware verification: Applying formal methods to Intel® TDX Module. In: Proc. TACAS. LNCS 16506, Springer (2026)
28. Beyer, D., Chien, P.C., Huang, B., Lee, N.Z., Lemberger, T.: The Intel TDX Module benchmark set. Zenodo (2025). https://doi.org/10.5281/zenodo.16547223
29. Beyer, D., Chien, P.C., Jankola, M.: BenchCloud: A platform for scalable performance benchmarking. In: Proc. ASE. pp. 2386–2389. ACM (2024). https://doi.org/10.1145/3691620.3695358
30. 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
31. Beyer, D., Hartmanns, A., Kordon, F.: TOOLympics Challenge 2023: Updates, Results, Successes of the Formal-Methods Competitions. LNCS 14550, Springer (2024). https://doi.org/10.1007/978-3-031-67695-6
32. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
33. 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

34. 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
35. 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
36. 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
37. Beyer, D., Wachowitz, H.: FM-Weck: Containerized execution of formal-methods tools. In: Proc. FM. pp. 39–47. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_3
38. Beyer, D., Wachowitz, H.: Fm-weck Release 1.6.0. Zenodo (2026). https://doi.org/10.5281/zenodo.18650812
39. Bürdek, J., Lochau, M., Bauregger, S., Holzer, A., von Rhein, A., Apel, S., Beyer, D.: Facilitating reuse in multi-goal test-suite generation for software product lines. In: Proc. FASE. pp. 84–99. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_6
40. 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)
41. Cadar, C., Nowack, M.: Klee symbolic execution engine in 2019 (competition contribution). Int. J. Softw. Tools Technol. Transf. **23**(6), 867 – 870 (December 2021). https://doi.org/10.1007/s10009-020-00570-3
42. Chalupa, M., Novák, J., Strejček, J.: Symbiotic 8: Parallel and targeted test generation (competition contribution). In: Proc. FASE. pp. 368–372. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_20
43. Chalupa, M., Strejček, J., Vitovská, M.: Joint forces for memory safety checking. In: Proc. SPIN. pp. 115–132. Springer (2018). https://doi.org/10.1007/978-3-319-94111-0_7
44. Cok, D.R., Déharbe, D., Weber, T.: The 2014 SMT competition. JSAT **9**, 207–242 (2016)
45. Dutta, A., Maghareh, R., Jaffar, J., Godboley, S., Yu, X.L.: TracerX: Pruning dynamic symbolic execution with deletion and weakest precondition interpolation (competition contribution). In: Proc. FASE. pp. 320–325. LNCS 14573, Springer (2024). https://doi.org/10.1007/978-3-031-57259-3_19
46. Ergang, M., Trtík, M.: TestCoCa: Test-suite coverage calculator (competition contribution). In: Proc. FASE. LNCS 16504, Springer (2026)
47. 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
48. 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
49. 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
50. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Softw. Eng. **30**(1), 3–16 (2004). https://doi.org/10.1109/TSE.2004.1265732

51. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Proc. ASE. pp. 407–416. ACM (2010). https://doi.org/10.1145/1858996.1859084
52. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: Tracer: A symbolic execution tool for verification. In: Proc. CAV. pp. 758–766. LNCS 7358, Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_61
53. Jakobs, M.C., Richter, C.: CoVeriTest with adaptive time scheduling (competition contribution). In: Proc. FASE. pp. 358–362. LNCS 12649, Springer (2021). https://doi.org/10.1007/978-3-030-71500-7_18
54. Jonáš, M., Strejček, J., Trtík, M.: Fizzer with local space fuzzing (competition contribution). In: Proc. FASE. pp. 275–280. LNCS 15693, Springer (2025). https://doi.org/10.1007/978-3-031-90900-9_14
55. Jonáš, M., Strejček, J., Trtík, M., Urban, L.: Fizzer: New gray-box fuzzer (competition contribution). In: Proc. FASE. pp. 309–313. LNCS 14573, Springer (2024). https://doi.org/10.1007/978-3-031-57259-3_17
56. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
57. 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
58. Lemberger, T., Wachowitz, H.: AFL-TC: Transforming fuzzer test inputs for Test-Comp (competition contribution). In: Proc. FASE. LNCS 16504, Springer (2026)
59. Marques, F., Santos, J.F., Santos, N., Adão, P.: Concolic execution for webassembly (artifact). Dagstuhl Artifacts Series **8**(2), 20:1–20:3 (2022). https://doi.org/10.4230/DARTS.8.2.20
60. Meudec, C.: Sikraken: Symbolic execution using constraint logic programming for generating test inputs (competition contribution). In: Proc. FASE. LNCS 16504, Springer (2026)
61. Misonizhnik, A., Morozov, S., Kostyukov, Y., Kalugin, V., Babushkin, A., Mordvinov, D., Ivanov, D.: KLEEF: Symbolic execution engine (competition contribution). In: Proc. FASE. pp. 314–319. LNCS 14573, Springer (2024). https://doi.org/10.1007/978-3-031-57259-3_18
62. Panichella, S., Gambi, A., Zampetti, F., Riccio, V.: SBST tool competition 2021. In: Proc. SBST. pp. 20–27. IEEE (2021). https://doi.org/10.1109/SBST52555.2021.00011
63. Pavlidou, A.: Zenodo introduces a new way to manage and correct published records (December 2025), available online, accessed 2026-02-21
64. Ruland, S., Lochau, M., Jakobs, M.C.: HybridTiger: Hybrid model checking and domination-based partitioning for efficient multi-goal test-suite generation (competition contribution). In: Proc. FASE. pp. 520–524. LNCS 12076, Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_26
65. Song, J., Alves-Foss, J.: The DARPA cyber grand challenge: A competitor's perspective, part 2. IEEE Security and Privacy **14**(1), 76–81 (2016). https://doi.org/10.1109/MSP.2016.14
66. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: Proc. IJCAR, pp. 367–373. LNCS 8562, Springer (2014). https://doi.org/10.1007/978-3-319-08587-6_28
67. Sutcliffe, G.: The CADE ATP system competition: CASC. AI Magazine **37**(2), 99–101 (2016). https://doi.org/10.1609/aimag.v37i2.2620
68. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test-input generation with Java PathFinder. In: Proc. ISSTA. pp. 97–107. ACM (2004). https://doi.org/10.1145/1007512.1007526

69. Wei, C., Wu, T., Menezes, R.S., Shmarov, F., Aljaafari, F., Godboley, S., Alshmrany, K., de Freitas, R., Cordeiro, L.: ESBMC v7.7: Automating branch-coverage analysis using CFG-based instrumentation and smt solving (competition contribution). In: Proc. FASE. pp. 281–286. LNCS 15693, Springer (2025). https://doi.org/10.1007/978-3-031-90900-9_15
70. Wendler, P., Beyer, D.: sosy-lab/benchexec: Release 3.34. Zenodo (2026). https://doi.org/10.5281/zenodo.18455156
71. Zhang, G., Chen, Z., Wang, J.: FDSE v2: Variable importance guided hybrid fuzzing (competition contribution). In: Proc. FASE. LNCS 16504, Springer (2026)
72. Zhang, G., Shuai, Z., Ma, K., Liu, K., Chen, Z., Wang, J.: FDSE: Enhance symbolic execution by fuzzing-based pre-analysis (competition contribution). In: Proc. FASE. pp. 304–308. LNCS 14573, Springer (2024). https://doi.org/10.1007/978-3-031-57259-3_16