




Evaluating Software Verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026)

Dirk Beyer ¹  and Jan Strejček ²

¹ LMU Munich, Munich, Germany

² Masaryk University, Brno, Czechia

Abstract. The *Competition on Software Verification (SV-COMP)* regularly evaluates software verifiers and witness validators processing programs in C and Java. SV-COMP 2026 evaluated 61 verifiers and 16 validators for C programs and 11 verifiers and 3 validators for Java programs. Out of these, 43 verifiers and 13 validators participated with an active support of teams led by 44 different representatives from 12 countries. The verification track of the competition was executed on a benchmark set of 36 402 verification tasks with C programs and 6 different specifications and 1 731 verification tasks with Java programs and 2 different specifications. The validation track analyzed 229 118 witnesses generated in the verification track for C programs and also 135 handcrafted witnesses. On top of that, SV-COMP 2026 considered also 254 verification tasks in the recently introduced format SV-LIB and evaluated 3 verifiers and 1 validator for this format. Moreover, there was also a demo category sponsored by Huawei, consisting of selected verification tasks with concurrent C programs. To keep our growing competition sustainable and up to date, we made several changes in its processes and settings.


Keywords: Formal Verification · Program Analysis · Competition · Software Verification · Verification Tasks · Verification Witnesses · Witness Validation · Benchmark · Specification · C Language · Java Language · SV-LIB · SV-COMP · SV-Benchmarks · BENCHEXEC · CoVeriTeam

1 Introduction

This report presents the objectives, processes, rules, participants, and results of the *15th Competition on Software Verification* (<https://sv-comp.sosy-lab.org/2026>) (SV-COMP 2026), which was again the largest comparative evaluation ever in this area. This paper extends the series of competition reports (see footnote)

This report extends the series [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 50] of previous reports on SV-COMP.

Reproduction packages are available on Zenodo (see Table 4).

 dirk.beyer@sosy-lab.org

documenting the growth and evolution of the competition over these 15 years. This year, we focus on the changes in the competition done in 2026, including the newly added categories and modifications in witness support.

The objectives of the competitions were discussed earlier (1–4 [26]) and extended over the years (5–6 [27] and 7 [31]):

1. provide an overview of the state of the art in software-verification technology and increase visibility of the most recent software verifiers,
2. establish a repository of software-verification tasks that is publicly available for free use as a standard benchmark suite for evaluating verification software,
3. establish standards that make it possible to compare different verification tools, including a property language and formats for the results,
4. accelerate the transfer of new verification technology to industrial practice by identifying the strengths of the various verifiers on a diverse set of tasks,
5. educate PhD students and others on performing reproducible benchmarking, packaging tools, and running robust and accurate research experiments,
6. provide research teams that do not have sufficient computing resources with the opportunity to obtain experimental results on large benchmark sets, and
7. conserve tools for formal methods for later reuse by using a standardized format to announce archives (via DOIs), default options, contacts, competition participation, and other meta data in a central repository.

The SV-COMP 2020 report [27] discusses the achievements of the SV-COMP competition so far with respect to these objectives.

Related Competitions. SV-COMP is one of many competitions that measure progress of research in the area of formal methods [18]. Competitions can lead to fair and accurate comparative evaluations because of the involvement of the developing teams. The competitions most related to SV-COMP are RERS [98], VerifyThis [80], Test-Comp [32], and TermCOMP [90]. The SV-COMP 2020 report [27] provides a more detailed discussion.

Quick Summary of Changes. We aim to keep the setup of the competition stable. On the other hand, the competition setup has to reflect the advances of the research area. The edition 2026 brought the following changes:

- The tool registration process and qualification criteria were simplified.
- Until 2025, the competition jury consists of representatives of the tools qualified for the competition. Since 2026, the jury contains representatives of the tools registered to the competition and their duties are slightly reduced.
- We introduced and applied systematic naming of categories. Each category now starts with the programming language of the contained programs (C, Java, or SV-LIB). The names of base categories then continue with the specification property of the contained verification tasks. We also renamed some properties to keep the names coherent. In particular, we renamed the properties `assert_java` and `runtime-exception` considered in Java benchmarks to `valid-assert` and `no-runtime-exception`, respectively.

- The meta category *FalsificationOverall* was renamed to *C.FalseOverall* and now it contains also benchmarks with the `termination` property. We introduced the dual meta category called *C.TrueOverall*.
- We had a demo category *C.Huawei-Concurrency-Challenges* sponsored by Huawei, consisting of selected tasks from the category *C.Concurrency*.
- The number of considered verification tasks again increased in both C and Java languages: the number of C tasks increased from 33 353 in 2025 to 36 402 and the number of Java tasks increased from 1 345 in 2025 (including tasks in demo categories) to 1 731.
- Besides benchmarks with programs in C and Java, we added a demo category with 254 verification tasks in the recently introduced format SV-LIB [40].
- We updated the support of witness formats. In particular, we now support the [witness format 2.1](#) and we discontinued support for correctness witnesses in format 1.0 (with the exception of witnesses for Java programs and the witnesses produced by verifiers without active team support).
- We added a new category *C.termination.ViolationWitnesses* of 17 handcrafted validation tasks with witnesses of program non-termination in format 2.1.
- The CPU time limit for validation of a correctness witness was lowered from 15 min to 5 min.
- We rewrote the scripts that post-process the output of all evaluated tools to make them more modular, efficient, and maintainable. The intermediate data are now stored in CSV instead of XML files.
- As ETAPS 2025 was held in Canada and many community members were not able to attend the traditional SV-COMP session there, we organized a one-day workshop on the current state and future development of SV-COMP and Test-Comp in April 2025 at Frauenchiemsee. Following its success, the second [SV-COMP/Test-Comp Workshop](#) was held on March 17, 2026 in Munich.

2 Organization and Processes

Organization. For the first 13 years, SV-COMP was ran by a single organizer (Dirk Beyer 2012–2017, Tomáš Vojnar 2018, Dirk Beyer 2019-2024). To handle its growing complexity, an organization committee was established before SV-COMP 2025. All members of the committee were also working on SV-COMP 2026. More precisely, the competition had two chairs (Dirk Beyer and Jan Strejček) and committee members in charge of benchmark quality (Zsófia Ádám, Raphaël Monat, Simmo Saan, and Frank Schüssele), category structure (Thomas Lemberger), infrastructure development (Philipp Wendler, Po-Chun Chien, Marek Jankola, Henrik Wachowitz, Matthias Kettl, and Marian Lingsch-Rosenfeld), and reproducibility (Levente Bajczi). For 2026, the committee was extended by members responsible for tool qualification (Paulína Ayaziová, Matthias Heizmann, Marian Lingsch-Rosenfeld, Felix Mächtle, Raphaël Monat, Malte Mues, and Jan-Niclas Serr).

Procedure. SV-COMP is an open competition (also known as comparative evaluation). Verification tasks and handcrafted validation tasks are publicly available in a repository ([Table 5](#)) where anyone can contribute. The competition has basically

two phases: *training* and *evaluation*. During the training phase, participating teams can repeatedly submit new versions of their tool. The organizers run the tool on relevant tasks and provide the results to the whole community. The participants can inspect the results, fix bugs in their tools and submit a new version, or report an issue with some tasks. The set of verification tasks and handcrafted validation tasks is *frozen* approximately two weeks before the the end of the training phase. In the evaluation phase, the tools are again executed on all relevant tasks and the participants are asked for an inspection of the results. They can challenge the validity of some tasks and suggest modifications of these tasks. Finally, the tasks modified after the freezing are excluded from the score computation (they are marked as *void*) and the results are announced on the competition web site.

Competition Jury. The competition jury reviews the competition contribution papers and helps the organizer with resolving any disputes that might occur. The jury consists of the SV-COMP chairs and one representative of each *registered* verifier and validator (a tool registered both as a verifier and a validator can have two representatives). In the previous editions, the jury consisted of the representatives of *qualified* tools and was responsible also for checking whether the tools registered to the next edition of SV-COMP fulfill the qualification criteria. Before this edition, we transferred this duty to the organization committee members mentioned above. The current representatives are listed in [Tables 6 and 7](#). The jury includes one additional member, A. R. Koçal (Technische Universität München, Germany), representing the tool [GOBLINT-PAR](#) which was registered, but not qualified to the competition. The current jury is also listed on the web site (<https://sv-comp.sosy-lab.org/2026/committee.php>).

3 Tasks, Workflow, and Scoring

Verification and Validation Tasks. A *verification task* consists of a program, a property to be verified, and the expected verification result. Tools for solving verification tasks are called *verifiers*. A *validation task* is a program, a property, and a witness of the program correctness or property violation to be validated. A typical validation task is a verification task extended with a witness generated by a verifier that solved the verification task. Some validation tasks (e.g., the handcrafted ones) contain also the expected validation result. Tools for solving validation tasks are called *validators*. SV-COMP 2026 used the [task-definition format in version 2.1](#) to denote the verification and validation tasks.

Programs and Properties. In connection with C programs, we considered 6 different properties: *unreachability of a given function* (referenced as `unreach-call` in the competition), *memory safety* (`valid-memsafety`) composed of three sub-properties saying that all pointer dereferences are valid (`valid-deref`), all memory deallocations are valid (`valid-free`), and all allocated memory is tracked (`valid-memtrack`), *memory cleanup* (`valid-memcleanup`) saying that all allocated memory is deallocated before the program terminates, *no overflow* (`no-overflow`) saying that operations on signed integers never overflow, *no data*

race (**no-data-race**) saying that the program does not contain any data race, and *termination* (**termination**) saying that the program always terminates. Note that a program is considered memory safe only if it satisfies all three subproperties. The subproperties are used in particular to report what is violated if a program is not memory safe. For Java programs, we considered the properties *assertion validity* (**valid-assert**) and *no runtime exception* (**no-runtime-exception**). In the previous years, these properties were referenced as **assert_java** and **runtime-exception**, respectively. We refer to the conference web page for precise definition of these properties (<https://sv-comp.sosy-lab.org/2026/rules.php>).

Additionally, SV-COMP 2026 also considered verification tasks in the recently introduced format for software-verification called SV-LIB [40]. One of the key features of the format is clear formal semantics, painfully missing for languages like C and Java. The format allows us to describe not only imperative programs, but also specifications and witnesses. The program specification is given by *tag annotations* which can describe a large class of safety and liveness properties. To check that an SV-LIB program satisfies its specification actually means to check that the tag annotations are correct. In SV-COMP, we call this property **correct-tags**.

Witnesses. SV-COMP 2026 supported several different formats for verification witnesses, depending on the programming language, considered property, specific program features, and verification result. In general, for correctness witnesses of C programs, we used witness formats 2.0 [8] and 2.1. The main benefit of the format 2.1 is added support of concurrent programs [83] and **termination** property [43, 154]. The format 2.1 allows correctness witnesses to contain *function contracts* and claim *inductivity* of invariants [95], but SV-COMP 2026 did not support these features as they were introduced only very shortly before the competition and did not have sufficient tool support. Note that after SV-COMP 2025, the competition abandoned the format 1.0 [37] for correctness witnesses of C programs with one exception: *inactive* verifiers (i.e., verifiers without active team support) can produce correctness witnesses in format 1.0 as in the previous year. For violation witnesses of C programs, we used witness formats 1.0, 2.0, and 2.1. For witnesses of Java programs, we used witness format 1.0. For SV-LIB tasks, we used witnesses in the SV-LIB 1.0 format. More information about supported witness formats for various classes of verification tasks is provided later.

Categories. The verification tasks are divided into *base categories* reflecting the programming language, the considered property, and loosely also some specific program features or the source of the benchmarks. Before SV-COMP 2026, we renamed all base categories such that now their names start with the prefix of the form $\langle \textit{language} \rangle. \langle \textit{property} \rangle$. Base categories are accumulated into *meta categories* whose names now also start with the prefix $\langle \textit{language} \rangle$.

For the C language, there are two levels of meta categories. The top level contains the categories *C.Overall*, *C.FalseOverall*, and *C.TrueOverall* of all C verification tasks. The category *C.FalseOverall* was previously called *FalsificationOverall* and contained only C tasks with safety properties (i.e., no **termination** property) The category shows bug-finding ability of participating tools by using a specific scoring schema explained below. The category *C.TrueOverall* was con-

sidered for the first time and it is dual to *C.FalseOverall*. Intuitively, its scoring schema shows ability to prove program correctness. One of the long-term goals of the SV-COMP community is to extend the benchmark set with more industrial verification task. As a step in this direction, the meta category *C.SoftwareSystems* was extended with one new base category *C.termination.SoftwareSystems-uthash*. Further, we improved and added new benchmarks to the category *C.unreach-call.SoftwareSystems-Intel-TDX-Module* introduced in the previous edition, such that now it contains 836 tasks compared to 146 in SV-COMP 2025. More details about this effort can be found in a case-study article [34], published in this volume as well. This improvement included introduction of a new competition-specific function `__VERIFIER_nondet_memory(void *, size_t)`, which havoc a given memory block (the first argument must be a valid pointer to the start of a memory block and the second argument must give the size). To increase the visibility and encourage support for this new function, more tasks to check the handling of this function will be introduced in the next edition. Finally, the meta category *C.ValidationCrafted* was extended with new base category *C.termination.ViolationWitnesses* of 17 crafted validation tasks with witnesses of `termination` violation in format 2.1.

For the Java language, the category with the `no-runtime-exception` property switched from demo category to regular base category. The overall number of Java verification tasks significantly increased from 1345 tasks in SV-COMP 2025 to 1731 tasks in SV-COMP 2026. One of the sources for extension was the tool ARGV, which can pull Java software projects from GitHub repositories and readily prepare them as benchmark set. The tool is also described in this volume [122]. The structure of all regular base and meta categories is shown in Fig. 1.

SV-COMP 2026 considered also some demo categories shown in Fig. 2. Four base demo categories accumulated in meta category *C.Huawei-Concurrency-Challenges* contain selected verification tasks from *C.Concurrency*. The meta category is sponsored by Huawei, who provided prize money for the top three verifiers in this category (excluding tools with at least one Huawei employee in the team, meta verifiers, and verifiers without active team support) in the amount of 3000 EUR for the winner, 1500 EUR for the second, and 500 EUR for the third place. The remaining three base demo categories accumulated in *SV-LIB.Overall* consist of SV-LIB tasks. They were treated as demo categories as they were announced very shortly before the competition deadlines.

The number of verification tasks in individual meta categories is shown in Tables 10, 11, and 12 presenting the results of the verification track. Finally, the competition web site (<https://sv-comp.sosy-lab.org/2026/benchmarks.php>) provides a detailed description of the categories.

Besides the verification track, the competition has also a validation track for validators of verification witnesses for C programs. We used the validation tasks with witnesses generated by verifiers on the verification tasks from *C.Overall* and the handcrafted validation tasks in *C.ValidationCrafted*. All tasks in the validation track were divided into 3 subtracks based on the witness type and used format, namely (a) correctness witnesses in formats 2.0 and 2.1, (b) violation

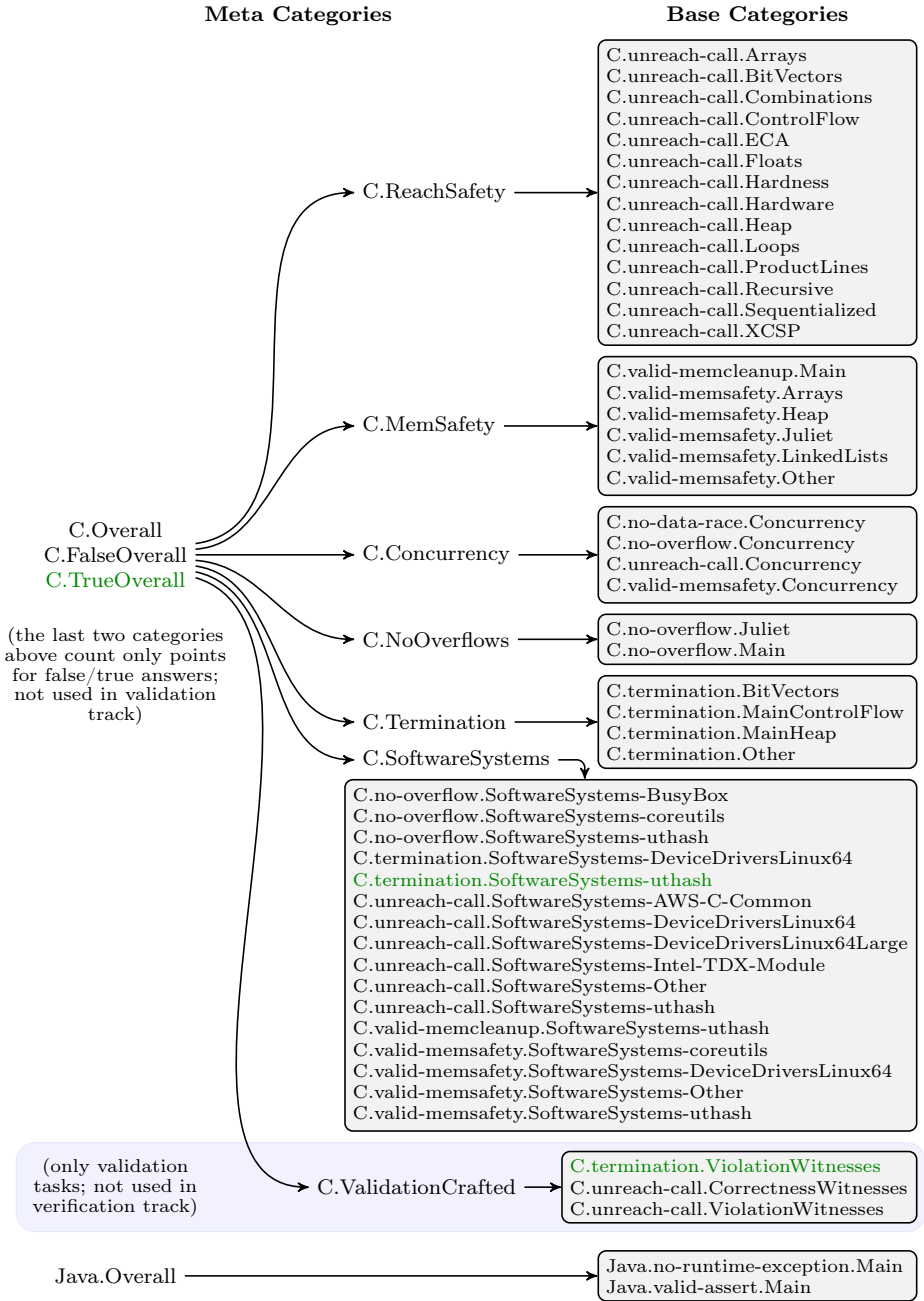


Fig. 1: Regular categories of SV-COMP 2026; new categories are green

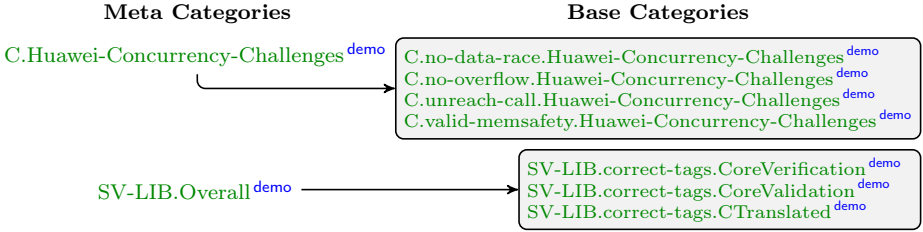


Fig. 2: Demo categories of SV-COMP 2026; all of them are new

Table 1: Supported witness formats in individual base categories; ‘-’ indicates that witnesses are not supported, ^{demo} means that the witnesses are supported only in *demo mode*, (1.0) denotes *legacy support* relevant only for inactive verifiers

Base Category		Correctness	Violation
Prefix	Suffix	Witnesses	Witnesses
<i>C.unreach-call.</i>	<i>Arrays, Floats, Heap</i>	–	1.0, 2.0, 2.1
	<i>Concurrency, Huawei*</i>	2.1 ^{demo}	1.0
	all others	(1.0), 2.0, 2.1	1.0, 2.0, 2.1
<i>C.valid-memsafety.</i>	<i>Concurrency, Huawei*</i>	–	1.0
	all others	–	1.0, 2.0*, 2.1*
<i>C.valid-memcleanup.</i>	all	–	1.0
<i>C.no-overflow.</i>	<i>Concurrency, Huawei*</i>	2.1 ^{demo}	1.0
	all others	(1.0), 2.0, 2.1	1.0, 2.0, 2.1
<i>C.no-data-race.</i>	all	–	1.0
<i>C.termination.</i>	all	2.1 ^{demo}	1.0, 2.1
<i>Java.valid-assert.</i>	<i>Main</i>	1.0 ^{demo}	1.0
<i>Java.no-runtime-exception.</i>	<i>Main</i>	–	1.0
<i>SV-LIB.correct-tags.</i>	all	SV-LIB	SV-LIB

* The formats 2.0 and 2.1 support only violation of the subproperties `valid-deref` and `valid-free` of `valid-memsafety`. If the violated subproperty is `valid-memtrack`, the format 1.0 has to be used.

witnesses in formats 2.0 and 2.1, and (c) violation witnesses in format 1.0. In each of these subtracks, the validation tasks were organized in the category structure corresponding to *C.Overall* and its subcategories, where each validation task generated by a verifier is in the base category determined by the corresponding verification task. Due to limitations of the witness formats and current validators, SV-COMP 2026 supported witnesses in individual formats only for selected properties and categories, as shown in Table 1. Consequently, some categories in some subtracks did not contain any validation task. The empty categories were removed from the category structure of each subtrack.

Table 1 clearly identifies several important research goals for the verification community: Develop and extend witness formats and validation tools for categories with insufficient validation support.

Competition Workflow. Roughly speaking, the inputs of the competition are *verification tasks* and, for each verifier and witness validator, the participating team creates (a) a *tool archive* on Zenodo from where the tool is downloaded, (b) a file `<tool>.yaml` at the *Formal-Methods Tools* repository with basic information about the tool, the name of its jury representative, identifier of the Zenodo archive, etc., (c) a *tool-info module* in the **BENCHEXEC** repository that specifies the interface for running the tool and interpreting its result, and (d) a merge request to the file `category-structure.yaml` in the *Benchmark Definitions* repository that declares the categories the tool participates in. The links to the mentioned repositories are in [Table 5](#).

The verification track proceeds as follows. The competition scripts run each verifier on all relevant verification tasks given by `category-structure.yaml`. If the verifier solves a task, it returns either `TRUE` meaning that the program satisfies the given property or `FALSE` meaning that the program violates the property. The output `TRUE` is accompanied with a *correctness witness* and the output `FALSE` with a *violation witnesses* in some of the formats supported by SV-COMP for the corresponding category (see [Table 1](#)). The table reflects the fact that the current witness infrastructure is not sufficiently developed to handle correctness witnesses of some verification tasks (e.g., tasks with C programs containing arrays, floats, heap manipulation, or parallelism, and tasks with properties `valid-memsafty` or `valid-memcleanup`). Correctness witnesses in some categories are thus not supported at all (denoted by ‘-’ in the table). Nevertheless, the situation is slowly improving. For example, the recently introduced witness format 2.1 supports parallel programs and `termination`. Because only very few validators can currently handle this format, correctness witnesses in some categories are supported only in *demo mode*. This means that witnesses can be generated and they are analyzed by available validators, but the results of the validation do not play any role in the score computation. The demo mode is also used for correctness witnesses of `Java.valid-assert.Main` due to a limited number of suitable validators. Further, there are some inactive verifiers of C programs that do not produce required correctness witnesses in format 2.0 or 2.1, but they can produce witnesses in format 1.0. To properly evaluate these verifiers, SV-COMP 2026 supports the format 1.0 in these cases. We emphasize that this *legacy support* is relevant only for inactive verifiers.

If there is no supported witness format or the only supported witness format is in demo mode, the witness generation is not required. Otherwise, a witness in some of the supported formats has to be generated. A witness in format 1.0 is written to file `witness.graphml`. A witness in format 2.0 or 2.1 is written to file `witness.yaml`. If a verifier generates both files, `witness.graphml` is ignored in the context of the verification track (it can be used in the validation track as explained below). We check whether (a) the generated witness is in a supported format, (b) its type corresponds to the verification result (correctness witnesses correspond to `TRUE`, violation witnesses to `FALSE`), and (c) it is syntactically valid. The last check is done by **WITNESSLINT** in case of C and by **PYSVLIB-LINTER**^{new} in case of SV-LIB. We skip the last check in case of Java as there is no witness linter

available. If any of the check fails and a witness was required, the verification task is seen as unsolved by the verifier. The last sentence does not apply to the tasks of demo category *C.Huawei-Concurrency-Challenges* where witnesses are not considered important. Checked witnesses are turned into validation tasks. Competition scripts then run all suitable validators on each validation task, where suitability is determined by `category-structure.yml`. Each validator can either confirm the task, refute it, or fail to solve it. A validator confirms a correctness witness by returning `TRUE` and refutes it by returning `FALSE`. A violation witness is confirmed by `FALSE` and refuted by `TRUE`.

The validation track uses the same inputs, only the verification tasks are replaced by *validation tasks*. These are the handcrafted validation tasks in *C.ValidationCrafted* and all validation tasks generated by verifiers on the verification tasks in *C.Overall* that passed the checks mentioned above. If a verifier generated two witness files `witness.graphml` and `witness.yml` and both of them pass the checks, we consider both in the validation track. A validation task contains an expected result if it is handcrafted or if it was generated with an incorrect verification result (i.e., it contains a correctness witness for a verification task with property violation or a violation witness for a verification task with a correct program). In the latter case, the expected validation result corresponds to refutation. Each validator is executed on all validation tasks of the categories it participates in according to `category-structure.yml`.

Computing Resources. The computing resources for each tool execution were the same as in SV-COMP 2025 with the exception of shorter CPU time limit for validation tasks with correctness witnesses. Each verifier run was limited by 15 GB of memory and 15 min of cumulative CPU time on 4 processing units. Each run of a validator was limited to 2 processing units, 7 GB of memory, and 1.5 min of CPU time for violation witnesses and 5 min of CPU time for correctness witnesses. The time limit for validation of correctness witnesses was 15 min in the previous years. The change should motivate verifiers to produce witnesses whose validation is substantially easier than solving the original verification tasks. The machines for running the experiments are part of a computer cluster at the SoSy-Lab at LMU, which consists of 168 machines, where each machine has one Intel Xeon E3-1230 v5 CPU with 8 processing units, 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). During the whole evaluation phase of the competition, approximately 1/3 of the cluster was not available due to a network switch failure. We used `BENCHEXEC` [45] to measure and control computing resources (CPU time, memory) and `BENCHCLOUD` [35] to distribute, install, run, and clean-up verification runs, and to collect the results.

Scoring Schema. For verification track, the scoring schema of SV-COMP 2026 in regular base categories was basically the one used since SV-COMP 2021. In [Table 2](#) we list all the cases when verification results are awarded with non-zero points, where a verification result is

Table 2: Scores per individual verification results used since SV-COMP 2021

Result	Points	Description
TRUE correct	+2	Program correctly reported to satisfy the property and the correctness witness confirmed (or not required)
TRUE incorrect	-32	Incorrect program reported as correct (wrong proof)
FALSE correct	+1	Property violation was correctly found and the violation witness was confirmed
FALSE incorrect	-16	Violation reported but the property holds (false alarm)

- *incorrect* if it does not agree with the expected result of the verification task,
- *correct* if it agrees with the expected result and the generated witness is confirmed by some validator. In the categories where correctness witnesses are not required, a verification results is *correct* whenever it agrees with the expected result.

When a verification result agrees with the expected result but no validator confirms the witness (although it is required), we call the result *correct-unconfirmed*. As *C.FalseOverall* was designed to compare the bug-finding ability of individual verifiers, the points in its base categories are awarded only for results FALSE (both correct and incorrect), while the results TRUE are ignored. SV-COMP 2026 introduced a dual category called *C.TrueOverall* designed to compare the ability of verifiers to prove program correctness. Hence, the points in its base categories are awarded only for results TRUE, while the results FALSE are ignored.

Slightly different rules were applied in base demo categories. In the base categories of *C.Huawei-Concurrency-Challenges*, witnesses are not considered important and thus they are always treated as confirmed. In the base categories of *SV-LIB.Overall*, every correctness witness checked by the linter is immediately treated as confirmed because there was no validator for these witnesses available.

The score of a verifier in a base category is simply the sum of the points for individual verification tasks. The score for a meta category is computed from the scores of all contained (meta or base) categories on the next level and the number of tasks in these categories. Formally, if a meta category contains k categories of the next level and the i -th contained category has the score s_i and consists of n_i verification tasks, then the meta category gets the score $(\sum_{i=1}^k s_i/n_i) \cdot (\sum_{i=1}^k n_i)/k$, i.e., the sum of scores in each category normalized by the number of tasks in the category multiplied by the average number of tasks in the contained categories. Note that until SV-COMP 2024, the numbers n_i included also void tasks that are technically in categories but are not used in the competition (a verification task is marked as *void* typically because it was changed for some serious reason after the task freezing deadline). Since SV-COMP 2025, these void tasks are not included in n_i .

For validation track, the scoring schema of SV-COMP 2026 in base categories was the same as in the previous editions. The biggest difference from the verification track comes from the fact that many validation tasks do not contain the expected

Table 3: Scores per individual validation results used since SV-COMP 2024

Result	Points	Description for correctness witnesses
		Description for violation witnesses
TRUE correct	+2	The correctness witness was correctly confirmed The violation witness was correctly refuted
TRUE incorrect	-32	The correctness witness was confirmed but it is not correct The violation witness was refuted but it is correct
FALSE correct	+1	The correctness witness was correctly refuted The violation witness was correctly confirmed
FALSE incorrect	-16	The correctness witness was refuted but it is correct The violation witness was confirmed but it is not correct

validation result. In fact, it is contained only in handcrafted validation tasks and in the tasks generated by verifiers that solve some verification task incorrectly (and thus the witness they produce should be refuted). For the remaining tasks, the expected results are determined by voting. For each such a task, we collect the results from all validators that solved (i.e., confirmed or refuted) the task. If we have at least two such results and at least 75 % of them agree on their decision, then the expected result is set by the majority vote. In all other cases, the expected result is not determined and the validation task has no influence on the validation track results as it is considered *void*. Tool developers can inspect the results and convince the community that a voted expected results is in fact wrong. In such a case, the corresponding validation task is removed from the competition.

Table 3 lists all the cases when validation results are awarded with non-zero points. A validation result TRUE or FALSE is considered *correct* if it agrees with the expected result and *incorrect* otherwise. To compute the scores in base categories, we virtually divide each base category C into two subcategories C_c, C_w , where

- C_c contains the witnesses that are expected to be *correct* (i.e., correctness witnesses with the expected result TRUE and violation witnesses with the expected result FALSE), and
- C_w contains the witnesses that are expected to be *wrong* (i.e., correctness witnesses with the expected result FALSE and violation witnesses with the expected result TRUE).

The score of a validator in each subcategory C_c, C_w is the sum of the points for individual validation tasks in the subcategory. If some of the subcategories C_c, C_w is empty, the score for C is directly the score for the non-empty subcategory. If both subcategories C_c, C_w are non-empty and have respective scores s_c, s_w , then the score for C is computed as for a meta category in the verification track, i.e., $(\frac{s_c}{|C_c|} + \frac{s_w}{|C_w|}) \cdot \frac{|C_c| + |C_w|}{2}$. The score of a validator in a meta category is then computed by the same process as in the verification track. Note that all the scores presented in this paper and on the competition web are rounded to integers before printing, but their computation is done with a higher precision.

Table 4: Artifacts published for SV-COMP 2026

Content	DOI	Reference
Verification Tasks	10.5281/zenodo.18650775	[53]
Competition Results	10.5281/zenodo.18651757	[52]
Formal-Methods Tools	10.5281/zenodo.18650756	[33]
Verification Witnesses	10.5281/zenodo.18651735	[54]
Witness Format	10.5281/zenodo.17277275	[51]
<code>BENCHEXEC</code>	10.5281/zenodo.18455156	[151]
<code>FM-WECK</code>	10.5281/zenodo.18650812	[55]
<code>BENCHCLOUD</code>	10.5281/zenodo.18670174	[16]

Table 5: Publicly available components for reproducing SV-COMP 2026

Component	Repository	Version
Verification Tasks	gitlab.com/sosy-lab/benchmarking/sv-benchmarks	svcomp26
Benchmark Definitions	gitlab.com/sosy-lab/sv-comp/bench-defs	svcomp26
Formal-Methods Tools	gitlab.com/sosy-lab/benchmarking/fm-tools	2.3
<code>BENCHEXEC</code> (Benchmarking)	github.com/sosy-lab/benchexec	3.34
<code>BENCHCLOUD</code> (Distribution)	gitlab.com/sosy-lab/software/benchcloud	1.5.0
Witness Format	gitlab.com/sosy-lab/benchmarking/sv-witnesses	2.1.2
<code>FM-WECK</code> for CI	gitlab.com/sosy-lab/software/fm-weck	1.6.0
Processing Scripts	gitlab.com/sosy-lab/benchmarking/competition-scripts	svcomp26

Ranking and Medals. The rank of a verifier in each category was decided based on the achieved score. In case of a tie, we used the *success run time* as the secondary criterion, which is the total CPU time of the verifier over the tasks in a given category for which the verifier received positive points. Ranking in validation track works in the same way. We recall that inactive tools and *meta verifiers* (defined in the next section) are excluded from official rankings. In *C.Huawei-Concurrency-Challenges*, the tools with at least one Huawei employee in the team are excluded from the rankings too. In each meta category except demo categories and the meta categories contained in *C.FalseOverall* and *C.TrueOverall*, we assign medals to the three top-ranking tools with a positive score. Huawei provided prize money to the three top-ranking verifiers in the category *C.Huawei-Concurrency-Challenges*.

Reproducibility. SV-COMP results must be reproducible. Therefore, all competition artifacts are published at Zenodo (see Table 4) with the relevant tools and data to guarantee their long-term availability and immutability. Further, all major components are maintained in public version-control repositories. In Table 5 we list these components with the links to the repositories and their versions used in SV-COMP 2026. Most of these components are described with more details in the SV-COMP 2016 report [24]. Later, `BENCHCLOUD` was introduced to distribute the benchmarking jobs in an elastic cloud and collect results. Moreover, `FM-WECK` was used to continuously check (GitLab CI pipeline)

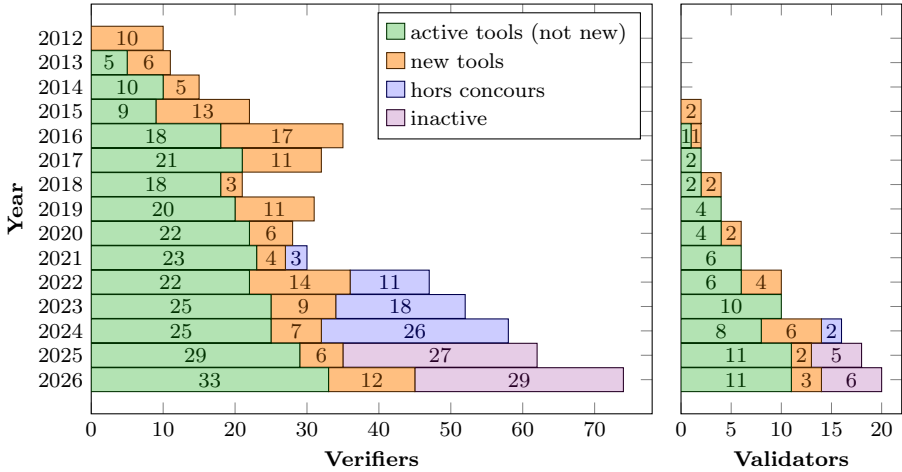


Fig. 3: Number of evaluated verifiers and validators in each year of SV-COMP; three new hors concours tools of 2022 counted only as new, not as hors concours

whether tools can be executed in the competition environment. The processing scripts to execute the experiments and post-process the data into tables, scores, and rankings are also publicly released.

For the reproducibility reasons, SV-COMP requires that the verifiers and validators must be publicly available for download and has a license that (a) allows reproduction and evaluation by anybody (including publication of results) and (b) does not restrict the usage of the verifier output (log files, witnesses). The verification and handcrafted validation tasks used in the competition are also accompanied by a license. In this case, the stated license must allow to

- view, understand, investigate, and reverse engineer the algorithm or system,
- change the program (in particular, pre-process and adopt the programs to be useful for a verification task),
- (re-)distribute the (original and changed) program under the same terms (in particular, in replication packages for research projects or as regression tests),
- compile and execute the program (in particular, for the purpose of verifying that a specification violation exists),
- and commercially take advantage of the program (in particular, to not exclude developers of commercial verifiers).

4 Participating Verifiers and Validators

In total, SV-COMP 2026 evaluates 74 verification and 20 validation tools. Besides 45 verifiers and 14 validators registered to and supported in the competition by development teams, we also evaluated some tools participating in previous years but not actively registered and supported this year (Fig. 3). These tools are called *inactive*, clearly marked with \emptyset in all tables, and they do not appear in

rankings. Further, we clearly distinguish *meta verifiers* according to the following characterization approved by the community of SV-COMP 2023.

A *meta verifier* is a combination of at least two existing verification components such that each result produced by the combination can be computed by some of its components alone. A verifier is the result of research and engineering in verification algorithms and approaches, while a typical meta verifier selects a verification component to run, sets up its parameters, and potentially post-processes its output.

Meta verifiers are annotated with meta in all tables and also excluded from rankings. Note that before SV-COMP 2025, both inactive tools and meta verifiers were marked as *hors-concours* participants and not properly distinguished. [Figure 3](#) shows the evolution of the number of verifiers and validators participating in individual editions of SV-COMP.

[Tables 6](#) and [7](#) provide the list of all verifiers and validators evaluated in SV-COMP 2026, respectively. The tables contain the tool name (with hyperlink),

Table 6: Participating verifiers with tool references, representing jury members and their affiliations; ∅ for inactive, meta for meta verifiers, and new for first-time participants

Verifier	Ref.	Jury Member	Affiliation
2LS ∅	[59, 118]	–	–
AISE	[113, 150]	Z. Chen	NUDT, China
APROVE	[114]	N. Lommen	RWTH Aachen, Germany
BRICK	[60]	L. Bu	Nanjing U., China
BUBAAK ∅	[62, 64]	–	–
BUBAAK-SPLit ∅	[63]	–	–
CBMC ∅	[69, 108]	–	–
COASTAL ∅	[147]	–	–
CoOPERACE meta		V. Vojdani	U. Tartu, Estonia
CPACHECKER	[12, 13]	M. Jankola	LMU Munich, Germany
CPALOCKATOR ∅	[5, 6]	–	–
CPA-BAM-BnB ∅	[4, 149]	–	–
CPA-BAM-SMG ∅		–	–
CPV	[68]	P.-C. Chien	LMU Munich, Germany
CRUX ∅	[82, 141]	–	–
CSEQ	[74, 103]	O. Inverso	Gran Sasso Sc. I., Italy
DARTAGNAN	[89, 112]	H. Ponce de León	Huawei Dresden, Germany
DASA new	[130]	F. Mächtle	U. Lübeck, Germany
DEAGLE	[93]	F. He	Tsinghua U., China
DIVINE ∅	[15, 109]	–	–
EBF ∅	[3]	–	–
EMERGENThETA	[124, 125]	L. Bajczi	BME Budapest, Hungary
ESBMC-INCR	[70, 73]	X. Li	U. Manchester, UK

(continues on next page)

Table 6: Participating verifiers (continued)

Verifier	Ref.	Jury Member	Affiliation
ESBMC-KIND	[88, 152]	X. Li	U. Manchester, UK
FRAMA-C-SV [∅]	[47, 75]	–	–
GAZER-THETA [∅]	[1, 92]	–	–
GDART	[128]	M. Mues	U. Wuppertal, Germany
GDART-LLVM [∅]		–	–
GOBLINT	[140, 148]	S. Saan	U. Tartu, Estonia
GOBLITCH ^{new}	[97]	K. Holter	U. Tartu, Estonia
GRAVES-CPA ^{∅ meta}	[110]	–	–
HORNIX	[57]	M. Blicha	U. Lugano, Switzerland
IEKKE ^{new}	[56]	P. Di Biase	Unimol, Italy
INFER [∅]	[61, 106]	–	–
JAVA-RANGER [∅]	[100, 143]	–	–
JAYHORN	[105, 142]	H. Mousavi	U. Tehran and TIAS, Iran
JBMC	[71, 72]	P. Schrammel	Diffblue, UK
JDART [∅]	[117, 127]	–	–
JLiSA ^{new}	[7]	G. Zanatta	U. Venice, Italy
KORN	[85, 86]	G. Ernst	LMU Munich, Germany
LAZY-CSEQ [∅]	[101, 102]	–	–
LF-CHECKER [∅]		–	–
LOCKSMITH [∅]	[135]	–	–
MLB		L. Bu	Nanjing U., China
MOPSA	[104, 121]	R. Monat	Inria & U. Lille, France
MuVAL ^{new}		H. Unno	Tohoku U., Japan
NACPA ^{meta}	[111]	H. Wachowitz	LMU Munich, Germany
OGCHECKER ^{new}		Z. Yang	Xidian U., China
PeSCo-CPA ^{∅ meta}	[137, 138]	–	–
PICHECKER [∅]	[144]	–	–
PINAKA [∅]	[66]	–	–
PREDATORHP [∅]	[96, 133]	–	–
PROTON	[120, 129]	R. Metta	TCS, India
PySvLib-CHC ^{new}		G. Ernst	LMU Munich, Germany
RACERF	[76, 77]	T. Dacík	BUT, Czechia
REFUNCTION ^{new}	[126]	N. Moussaoui Remil	Inria and ÉNS, France
RE3VER ^{new}	[155]	A. Štěpková	Masaryk U., Brno, Czechia
SEAL ^{new}	[58]	T. Dacík	Brno U. Techn., Czechia
SPF [∅]	[131, 136]	–	–
SVF-SVC	[119]	M. Richards	U. New South Wales, AU
SvLibChecker ^{new}		M. Lingsch-Rosenfeld	LMU Munich, Germany
SV-SANITIZERS		S. Saan	U. Tartu, Estonia
SWAT	[115, 116]	N. Loose	U. Luebeck, Germany
SYMBIOTIC	[9, 65]	M. Jonáš	Masaryk U., Czechia
THETA	[145, 146]	C. Telbisz	BME Budapest, Hungary
THORN		L. Bajczi	BME Budapest, Hungary
UAUTOMIZER	[19, 94]	M. Heizmann	U. Freiburg, Germany
UGEMCUTTER	[87, 107]	D. Klumpp	U. Freiburg, Germany
UKOJAK	[84, 132]	M. Bentele	U. Freiburg, Germany

(continues on next page)

Table 6: Participating verifiers (continued)

Verifier	Ref.	Jury Member	Affiliation
UPARALIZER ^{new}	[17]	M. Barth	LMU Munich, Germany
UTAIPAN	[81, 91]	D. Dietsch	U. Freiburg, Germany
VERIABS [∅]	[2, 78]	–	–
VERIABSL [∅]	[79]	–	–
VERIOVER [∅]	–	–	–

references to papers that describe the tool, the representing jury member and the affiliation. The listings are also available on the competition web site at <https://sv-comp.sosy-lab.org/2026/systems.php>. Table 7 additionally indicates witness formats and types supported by individual validators. The support of correctness witnesses in format 1.0 for C programs is indicated by ‘(✓)’ to emphasize that it is used in the competition only because of the legacy support of these

Table 7: Participating validators with tool references, representing jury members, their affiliations, and indication of relevant supported witnesses depending on format version and type; [∅] for inactive, ^{new} for first-time participants, *Cor.* for correctness witnesses, *Vio.* for violation witnesses, (✓) for legacy support, and ✓ for newly added support; validator `PySvLib-Valid`.^{new} supports violation witnesses in SV-LIB format

Validator	Ref.	Jury Member	Affiliation	Witness Format			
				1.0		2.0 or 2.1	
				Cor.	Vio.	Cor.	Vio.
CONCURRENTW2T	[14]	Z. Ádám	BME Budapest		✓		
CPACHECKER	[41, 44]	M. Lingsch-Rosenfeld	LMU Munich	(✓)	✓	✓	✓
CPA-W2T [∅]	[37, 39]	–	–		✓		
CProver-w2t [∅]	[37, 39]	–	–		✓		
DARTAGNAN	[134]	H. Ponce de León	Huawei Dresden		✓		
GOBLINT	[139]	S. Saan	U. Tartu				✓
GWIT [∅]	[99]	–	–		✓		
JCWIT [∅]	[67]	–	–	✓			
LIV	[48]	M. Lingsch-Rosenfeld	LMU Munich	(✓)		✓	
METAVAL	[46]	M. Lingsch-Rosenfeld	LMU Munich	(✓)	✓ [∅]	✓	✓
MOPSA	[123]	R. Monat	Inria & U. Lille			✓	
NITWIT [∅]	[156]	–	–		✓		
SYMBIOTIC-WITCH [∅]	[10]	–	–		✓		
THETA ^{new}		L. Bajczi	BME Budapest				✓
UAUTOMIZER	[36, 38]	M. Ebbinghaus	U. Freiburg	(✓)	✓	✓	✓
UGEMCUTTER ^{new}		D. Klumpp	U. Freiburg			✓	
UREFEREE		F. Schüssele	U. Freiburg	(✓)		✓	
WIT4JAVA	[153]	T. Wu	U. Manchester		✓		
WITCH	[8, 11]	P. Ayaziová	Masaryk U., Brno				✓
PySvLib-Valid. ^{new}		M. Lingsch-Rosenfeld	LMU Munich			(supports SV-LIB)	

witnesses. Note that to check these witnesses, we actually used the version of the validators participating in SV-COMP 2025 when these witnesses were still officially supported. Further, we used [METAVAL](#) in the version participating in SV-COMP 2025 to check violation witnesses in format 1.0 because in SV-COMP 2026, the tool was registered only to the other two validation subtracks. Hence, ‘✓[∅]’ denotes that [METAVAL](#) participates in the corresponding subtrack as an inactive tool. [Table 8](#) lists the algorithms and techniques used by the verification and validation tools (some less frequent techniques were omitted due to limited space).

Table 8: Selected algorithms and techniques used by the participating tools; [∅] for inactive, ^{meta} for meta verifiers, and ^{new} for first-time participants

Tool	Abstract Interpretation	Algorithm Selection	ARG-Based Analysis	Automata-Based Analysis	Bit-Precise Analysis	Bounded Model Checking	CEGAR	Concurrency Support	Constrained Horn Clauses	Explicit-Value Analysis	Floating-Point Arithmetics	Interpolation	k-Induction	Lazy Abstraction	Numeric Interval Analysis	Portfolio	Predicate Abstraction	Property-Directed Reach.	Ranking Functions	Separation Logic	Shape Analysis	Symbolic Execution	Targeted Input Generation	Task Translation
AISE																								
AProVE																								
BRICK						✓	✓	✓							✓									
BUBAAK [∅]						✓	✓	✓																
BUBAAK-SPLit [∅]		✓		✓	✓	✓	✓	✓					✓			✓								
CBMC [∅]					✓	✓		✓																
COASTAL [∅]																								
CONCURRENTW2T																								
CoOPERACE ^{meta}		✓						✓																
CPA-BAM-BnB [∅]			✓							✓														
CPACHECKER		✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CPALOCKATOR [∅]			✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CPA-w2T [∅]			✓	✓																				
CPV		✓			✓	✓	✓	✓				✓	✓			✓	✓	✓	✓					✓
CRUX [∅]																						✓		
CSeq					✓	✓		✓																
DARTAGNAN					✓	✓		✓																
DASA ^{new}																							✓	
DEAGLE						✓		✓																
DIVINE [∅]		✓			✓	✓	✓	✓	✓							✓						✓		
EBF [∅]					✓	✓	✓	✓																
EMERGENThETA		✓	✓		✓	✓	✓	✓			✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ESBMC-INCR					✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ESBMC-KIND					✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
FRAMA-C-SV [∅]									✓	✓					✓									
CProver-w2T [∅]						✓																		

(continues on next page)

Table 8: Algorithms and techniques (continued)

Tool	Abstract Interpretation	Algorithm Selection	ARG-Based Analysis	Automata-Based Analysis	Bit-Precise Analysis	Bounded Model Checking	CEGAR	Concurrency Support	Constrained Horn Clauses	Explicit-Value Analysis	Floating-Point Arithmetics	Interpolation	k-Induction	Lazy Abstraction	Numeric Interval Analysis	Portfolio	Predicate Abstraction	Property-Directed Reach.	Ranking Functions	Separation Logic	Shape Analysis	Symbolic Execution	Targeted Input Generation	Task Translation
REFUNCTION ^{new}															✓									
GAZER-THETA [⊗]			✓						✓			✓		✓		✓	✓							
GDART					✓	✓										✓	✓							
GDART-LLVM [⊗]					✓	✓										✓	✓						✓	✓
GOBLINT	✓	✓			✓			✓		✓	✓				✓	✓								
GOBLINT-PAR		✓						✓		✓	✓				✓	✓								
GOBLITCH ^{new}	✓	✓								✓	✓				✓	✓							✓	
GRAVES-CPA ^{⊗ meta}		✓	✓		✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	
GWIT [⊗]					✓											✓							✓	
HORNIX								✓										✓						
IEKKE ^{new}						✓		✓																
INFER [⊗]															✓	✓				✓	✓			
JAVA-RANGER [⊗]					✓										✓	✓					✓	✓		
JAYHORN							✓		✓			✓		✓	✓	✓	✓							
JBMC					✓	✓		✓						✓	✓			✓						
JCWIT [⊗]						✓											✓						✓	
JDART [⊗]					✓												✓						✓	
JLISA ^{new}	✓																							
KORN							✓		✓	✓						✓	✓					✓	✓	
LAZY-CSEQ [⊗]					✓	✓		✓																
LIV																								✓
LOCKSMITH [⊗]								✓																
METAVAL		✓																						✓
MLB					✓											✓	✓					✓		
MOPSA	✓	✓							✓	✓					✓	✓							✓	
MUVAL ^{new}																			✓					
NACPA ^{meta}		✓														✓								
NITWIT [⊗]									✓								✓							
OGCHECKER ^{new}			✓					✓																
PESCO-CPA ^{⊗ meta}		✓	✓		✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	
PICHECKER [⊗]		✓	✓		✓	✓	✓	✓				✓					✓							
PINAKA [⊗]					✓	✓																	✓	
PREDATORHP [⊗]																					✓			
PROTON						✓																		
PySVLIB-CHC ^{new}								✓																
RACERF								✓																
RE3VER ^{new}					✓			✓			✓		✓		✓	✓					✓	✓	✓	

(continues on next page)

Table 8: Algorithms and techniques (continued)

Tool	Abstract Interpretation	Algorithm Selection	ARG-Based Analysis	Automata-Based Analysis	Bit-Precise Analysis	Bounded Model Checking	CEGAR	Concurrency Support	Constrained Horn Clauses	Explicit-Value Analysis	Floating-Point Arithmetics	Interpolation	k-Induction	Lazy Abstraction	Numeric Interval Analysis	Portfolio	Predicate Abstraction	Property-Directed Reach.	Ranking Functions	Separation Logic	Shape Analysis	Symbolic Execution	Targeted Input Generation	Task Translation
SEAL ^{new}	✓																							
SPF [⊗]								✓																
SVF-SVC																								
SVLIBCHECKER ^{new}			✓	✓		✓	✓		✓		✓	✓	✓				✓					✓	✓	✓
SV-SANITIZERS								✓																
SWAT																								
SYMBIOTIC					✓			✓			✓	✓	✓			✓	✓					✓	✓	✓
SYMBIOTIC-WITCH [⊗]																						✓	✓	✓
THETA	✓	✓			✓	✓	✓	✓	✓		✓	✓	✓			✓	✓							✓
THORN	✓							✓									✓	✓						
2LS [⊗]					✓	✓							✓											
UAUTOMIZER		✓		✓	✓	✓	✓	✓				✓	✓			✓	✓		✓					
UGEMCUTTER		✓		✓	✓	✓	✓	✓				✓	✓			✓	✓		✓					
UKOJAK				✓	✓	✓	✓	✓				✓	✓			✓	✓		✓					
UPARALIZER ^{new}				✓	✓	✓	✓	✓				✓	✓			✓	✓		✓					
UREFEREE		✓		✓	✓	✓	✓	✓				✓	✓			✓	✓		✓					
UTAIPAN		✓		✓	✓	✓	✓	✓	✓			✓	✓			✓	✓		✓					
VERIABS [⊗]		✓							✓				✓			✓	✓							
VERIABSL [⊗]		✓				✓	✓		✓				✓			✓	✓							
WITCH																						✓		
WIT4JAVA																							✓	

Table 9 gives an overview of common solver libraries and frameworks used by these tools. Note that both tables are based on information provided by teams that registered individual tools to SV-COMP 2026 or to some previous editions in the case of currently inactive tools. In Tables 8 and 9, we omitted the tools that would have no checkmark in the coresponding row. The web site <https://fm-tools.sosy-lab.org> provides more information about all tools evaluated in SV-COMP and Test-Comp since 2023 in a uniform way.

5 Results

The results of the competition represent the state of the art of what can be achieved with fully automatic software-verification tools on the given benchmark set. We report the *effectiveness* (the number of verification tasks that can be solved and correctness of the results, as accumulated in the score) and the *efficiency*

Table 9: Solver libraries and frameworks used as components by at least two participating tools; \emptyset for inactive, *meta* for meta verifiers, and *new* for first-time participants

Tool	APRON	BITWUZLA	BOOLECTOR	CPACHECKER	CPROVER	CVC	ELDARICA	ESBMC	FRAMA-C	GLUCOSE	GOLEM	JAVASMT	JPF	KLEE	MATHSAT	MINISAT	SMTINTERPOL	SYMBIOTIC	ULTIMATE	Z3
AISE						✓														✓
APROVE (KoAT+LoAT)																				✓
BRICK																				✓
BUBAAK \emptyset																✓				✓
CBMC \emptyset					✓											✓				✓
COASTAL \emptyset													✓							
CPA-BAM-BNB \emptyset				✓								✓			✓					
CPA-BAM-SMG \emptyset				✓								✓			✓					
CPACHECKER	✓			✓								✓			✓					
CPALOCKATOR \emptyset				✓								✓			✓					
CPV		✓	✓			✓									✓	✓				
CRUX \emptyset																				✓
CSEQ					✓											✓				
DARTAGNAN											✓									
DEAGLE																✓				
EBF \emptyset								✓							✓					
EMERGENTHETA		✓	✓			✓						✓			✓		✓			✓
ESBMC-INCR								✓							✓					
ESBMC-KIND								✓							✓					
FRAMA-C-SV \emptyset									✓											
REFUNCTION <i>new</i>	✓																			
GDART						✓														✓
GDART-LLVM \emptyset																				✓
GOBLINT	✓																			
GOBLINT-PAR	✓																			
GOBLITCH <i>new</i>	✓													✓				✓		✓
GRAVES-CPA \emptyset <i>meta</i>				✓								✓			✓					
HORNIX											✓									
IEKKE <i>new</i>																✓				
JAVA-RANGER \emptyset													✓							
JBMC					✓				✓							✓				
JDART \emptyset						✓							✓							✓
KORN							✓				✓									✓
LAZY-CSEQ \emptyset					✓											✓				
MOPSA	✓																			
MUVAL <i>new</i>																				✓
NACPA <i>meta</i>				✓																
OGCHECKER <i>new</i>				✓																
PESCO-CPA \emptyset <i>meta</i>				✓								✓			✓					
PICHECKER \emptyset				✓								✓			✓		✓			

(continues on next page)

Table 9: Solver libraries and frameworks (continued)

Tool	APRON	BITWUZLA	BOOLECTOR	CPACHECKER	CPROVER	CVC	ELDARICA	ESBMC	FRAMA-C	GLUCOSE	GOLEM	JAVASMT	JPF	KLEE	MATHSAT	MINISAT	SMTINTERPOL	SYMBIOTIC	ULTIMATE	Z3	
PYSVLIB-CHC ^{new}							✓				✓									✓	
RACERF								✓													
RE3VER ^{new}														✓					✓	✓	
SEAL ^{new}								✓													
SPF [⊗]													✓								
SVLIBCHECKER ^{new}															✓						✓
SWAT																					✓
SYMBIOTIC															✓						✓
THETA		✓	✓			✓									✓						✓
2LS [⊗]					✓					✓							✓				✓
UAUTOMIZER						✓									✓						✓
UGEMCUTTER						✓									✓						✓
UKOJAK						✓									✓						✓
UPARALIZER ^{new}						✓									✓						✓
UREFEREE						✓									✓						✓
UTAIPAN						✓									✓						✓
VERIABS [⊗]				✓	✓											✓					✓
VERIABSL [⊗]				✓	✓											✓					✓
WITCH														✓							✓

(resource consumption in terms of CPU time). The results are presented in the same way as in last years, such that the improvements compared to the last years are easy to identify. The results presented in this report were provided to the participants in advance and their objections have been settled.

Consumed Resources. Before we present the competition results, we report some statistics to give an impression of the overall computation work. One complete execution of all verifiers in the competition consisted of 1 069 940 verification runs (each verifier runs on each verification task of the categories in which the verifier participates), consuming 9 years of CPU time (without validation). Witness validation required 24 million validation runs (each validator runs on each validation task of the categories in which the validator participates) consuming 8 years of CPU time. During the training phase of the competition, we executed 7.5 million verification runs consuming 53 years of CPU time and 95 million validation runs consuming 18 years of CPU time.

Verification track. Tables 10, 11, and 12 present the quantitative overview of all verifiers in all meta categories except the subcategories of *C.FalseOverall* and *C.TrueOverall*. We split the presentation into three tables, one for the verifiers of C programs that participate with an active team support (Table 10), one for the inactive verifiers of C programs (Table 11), and the last for all verifiers of Java programs and SV-LIB tasks (Table 12). The head row lists meta categories

Table 10: Overview of the results of all actively participating verifiers for C; empty cells indicate opt-outs, ^{meta} for meta verifiers, ^{new} for first-time participants

Verifier	C.ReachSafety 14375 tasks max. score 23331	C.MemSafety 4145 tasks max. score 6529	C.Concurrency 3124 tasks max. score 5656	C.NoOverflows 8218 tasks max. score 13281	C.Termination 2146 tasks max. score 3734	C.SoftwareSystems 4394 tasks max. score 7140	C.FalseOverall 36402 tasks max. score 12195	C.TrueOverall 36402 tasks max. score 48413	C.Overall 36402 tasks max. score 60609	C.Huawei-Con.-Chall. ^{demo} 66 tasks max. score 123
AISE										
APROVE					2077					
BRICK										
CoOPERACE ^{meta}										
CPACHECKER	12866	4646	2146	8474	1152	1496	7389	20589	27979	3
CPV	10626				1358					
CSEQ			-12677							-48
DARTAGNAN			3516							71
DEAGLE			4630							-11
EMERGENTHETA	4414	642	2636	1821	796	-18	1091	10400	11491	0
ESBMC-INCR			2435							-70
ESBMC-KIND	11037	3309	2410	9109	1074	14	5232	18732	23964	-70
GOBLINT	4664	2840	2555	8658	1614	457		22673	22673	34
GOBLITCH ^{new}	6552					611				
HORNIX										
IEKKE ^{new}			3428							-12
KORN										
MOPSA	4501	3165	0	9906	1571	2746	1442	20637	22079	0
MUVAL ^{new}					356					
NACPA ^{meta}	5959	3940	1578	7869	860	1477	4089	17538	21627	1
OGCHECKER ^{new}			131							0
PROTON					3368					
RACERF										
REFUNCTION ^{new}					294					
RE3VER ^{new}	2583									
SEAL ^{new}										
SV-SANITIZERS		878		837						
SVF-SVC	-9866	0				-346				
SYMBIOTIC	10788	4738	42	8476	1427	2614	7833	17638	25471	0
THETA	5252	695	2157	2586	796	-18	-329	11885	11556	0
THORN	738	12	1247	-174	583	69	187	4178	4365	0
UAUTOMIZER	8243	3957	2986	10831	2616	730	5835	25637	31472	0
UGEMCUTTER			3189							1
UKOJAK	6012	2982	0	8821	0	290	3468	10347	13815	0
UPARALIZER ^{new}	8279			11049						
UTAIPAN	7013	3517	2453	10645	0	404	4243	17044	21287	0

Table 11: Overview of the results of inactive verifiers for C; empty cells indicate opt-outs, \emptyset for inactive, *meta* for meta verifiers

Verifier	C.ReachSafety 14375 tasks max. score 23331	C.MemSafety 4145 tasks max. score 6529	C.Concurrency 3124 tasks max. score 5656	C.NoOverflows 8218 tasks max. score 13281	C.Termination 2146 tasks max. score 3734	C.SoftwareSystems 4394 tasks max. score 7140	C.FalseOverall 36402 tasks max. score 12195	C.TrueOverall 36402 tasks max. score 48413	C.Overall 36402 tasks max. score 60609	C.Huawei-Con.-Chall. <i>demo</i> 66 tasks max. score 123
2LS \emptyset	8100	631	0	6808	1607	25	3063	10883	13946	0
BUBAAK \emptyset	9071	3296	-191	6483	1332	2121	6362	13401	19763	0
BUBAAK-SPLIT \emptyset	7959	3283	-191	6472	1042	1996	6192	12083	18275	0
CBMC \emptyset	1896	1817	812	7173	1085	-1690	-2932	13998	11066	0
CPA-BAM-BnB \emptyset						-1577				
CPA-BAM-SMG \emptyset		3039				-3361				
CPALockator \emptyset			-3540							0
CRUX \emptyset	2821			602						
DIVINE \emptyset	6371	532	379	0	0	-2524	476	242	718	2
EBF \emptyset			351							-88
FRAMA-C-SV \emptyset				1579						
GAZER-THETA \emptyset										
GDART-LLVM \emptyset										
GRAVES-CPA <i>meta</i>	4252	2106	238	3329	-994	-362	-6603	11090	4487	-35
INFER \emptyset	-131447		-8297	-77663		-28819				-354
LAZY-CSEQ \emptyset			-14183							-48
LF-CHECKER \emptyset			440							-18
LOCKSMITH \emptyset										
PESCO-CPA <i>meta</i>	8085	2904	1483	8424	985	-3112	2414	12836	15249	-15
PICHECKER \emptyset			382							-18
PINAKA \emptyset	3818			641	878					
PREDATORHP \emptyset		4543								
VERIABS \emptyset	14056									
VERIABSL \emptyset	14404									
VERIOOVER \emptyset										

with the number of valid tasks and the maximal score for each category. The tools are listed in alphabetical order (in Table 12, verifiers for Java are before those for SV-LIB) and every table row lists the scores of one verifier. An empty table cell means that the verifier did not participate in the corresponding meta category. In Tables 10 and 12, we indicate the top three candidates in each (non-demo) category by formatting their scores in bold face and in larger font size. We recall that inactive tools and meta verifiers are excluded from rankings. More information (including interactive tables, quantile plots for every category,

Table 12: Overview of the results of all verifiers for Java and SV-LIB; \emptyset for inactive, ^{meta} for meta verifiers, ^{new} for first-time participants

Verifier	Java.Overall 1731 tasks max. score 2821	SV-LIB.Overall ^{demo} 261 tasks max. score 391
COASTAL \emptyset	-13809	
DASA ^{new}	210	
GDART	1470	
JAVA-RANGER \emptyset	1059	
JAYHORN	-2472	
JBMC	1561	
JDART \emptyset	-11255	
JLISA ^{new}	1311	
MLB	335	
SPF \emptyset	-4741	
SWAT	1291	
CPACHECKER		133
PYSVLIB-CHC ^{new}		78
SVLIBCHECKER ^{new}		165

and also the raw data in XML format) is available on the competition web site (<https://sv-comp.sosy-lab.org/2026/results>) and in the results artifact (see Table 4). Note that the results for subcategories of *C.FalseOverall* and *C.TrueOverall* are not explicitly presented neither in this report nor on the web due to their marginal significance. The results can be obtained from the detailed results of the corresponding categories in *C.Overall* presented on the web.

Table 13 shows the medalists for each (non-demo) meta category. The cumulative run time of the verifier on these tasks is presented in the column ‘CPU Time’. The column ‘Solved Tasks’ shows the number of tasks in the corresponding category for which the verifier received positive points. The column ‘Unconf. Tasks’ provides the number of tasks for which the verifier returned a correct answer, but the corresponding required witness was not confirmed by any validator. The columns ‘False Alarms’ and ‘Wrong Proofs’ provide the number of verification tasks for which the verifier returned incorrect FALSE and incorrect TRUE, respectively. Recall that categories *C.FalseOverall* and *C.TrueOverall* completely ignore TRUE and FALSE results, respectively. As a consequence, there cannot be any wrong proof in *C.FalseOverall* or false alarm in *C.TrueOverall*.

The results in *C.Huawei-Concurrency-Challenges* are available in Tables 10 and 11. Because the representing jury member of DARTAGNAN is a Huawei employee and DIVINE \emptyset is inactive, the prize money went to GOBLINT (3 000 EUR), CPACHECKER (1 500 EUR), and UGEMCUTTER (500 EUR).

Score-Based Quantile Functions. The community often present the results of the comparative evaluation in the form of score-based quantile functions [21, 45] because these visualizations are easy to understand and provide more infor-

Table 13: Verification: Overview of the medalists in each (non-demo) meta category; values for ‘CPU time’ in hours and rounded to two significant digits, ‘Solved Tasks’ shows the number of tasks for which the verifier got positive points, ‘Unconf. Tasks’ shows the number of tasks where the verifier returned the correct answer and the corresponding required witness was not confirmed

<i>Category</i>	<i>Rank</i>	<i>Verifier</i>	<i>Score</i>	<i>CPU Time</i>	<i>Solved Tasks</i>	<i>Unconf. Tasks</i>	<i>False Alarms</i>	<i>Wrong Proofs</i>
<i>C.ReachSafety</i> (14375 tasks, max. score 23331)								
1		CPACHECKER	12866	150	8 053	365	2	0
2		ESBMC-KIND	11037	57	7 273	2 456	0	0
3		SYMBIOTIC	10788	110	4 080	207	1	0
<i>C.MemSafety</i> (4145 tasks, max. score 6529)								
1		SYMBIOTIC	4738	3.9	3 834	11	0	2
2		CPACHECKER	4646	19	3 814	5	1	0
3		UAUTOMIZER	3957	35	2 226	80	0	0
<i>C.Concurrency</i> (3124 tasks, max. score 5656)								
1		DEAGLE	4630	3.1	2 513	19	1	3
2		DARTAGNAN	3516	19	2 052	43	3	1
3		IEKKE ^{new}	3428	2.3	2 473	216	44	22
<i>C.NoOverflows</i> (8218 tasks, max. score 13281)								
1		UPARALIZER^{new}	11049	88	6 700	22	0	0
2		UAUTOMIZER	10831	67	6 669	59	1	0
3		UTAIPAN	10645	75	6 572	56	0	0
<i>C.Termination</i> (2146 tasks, max. score 3734)								
1		PROTON	3368	22	1 780	162	1	0
2		UAUTOMIZER	2616	17	1 558	5	0	0
3		APROVE	2077	31	1 011	42	0	0
<i>C.SoftwareSystems</i> (4394 tasks, max. score 7140)								
1		MOPSA	2746	11	1 920	28	0	0
2		SYMBIOTIC	2614	6.2	1 165	94	1	0
3		CPACHECKER	1496	28	1 857	257	0	0
<i>C.FalseOverall</i> (36402 tasks, max. score 12195)								
1		SYMBIOTIC	7833	42	7 378	52	2	
2		CPACHECKER	7389	100	7 668	454	4	
3		UAUTOMIZER	5835	53	4 864	569	7	
<i>C.TrueOverall</i> (36402 tasks, max. score 48413)								
1		UAUTOMIZER	25637	200	11 686	167		3
2		GOBLINT	22673	24	10 364	352		0
3		MOPSA	20637	26	9 766	26		0
<i>C.Overall</i> (36402 tasks, max. score 60609)								
1		UAUTOMIZER	31472	260	16 550	736	7	3
2		CPACHECKER	27979	240	21 632	930	4	0
3		SYMBIOTIC	25471	160	16 158	347	2	3
<i>Java.Overall</i> (1731 tasks, max. score 2821)								
1		JBMC	1561	1.5	1 033	315	1	13
2		GDART	1470	4.3	920	107	1	1
3		JLiSA ^{new}	1311	2.0	617	1	0	0

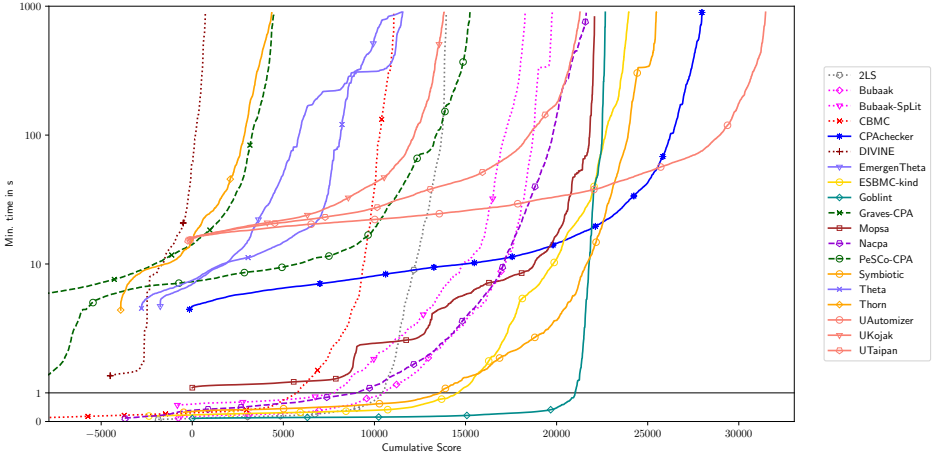


Fig. 4: Quantile functions for category *C.Overall*; each quantile function illustrates the quantile (x -coordinate) of the scores obtained by correct verification runs below a certain run time (y -coordinate), minus the overall penalty for incorrect results; more details were given previously [21]; a logarithmic scale is used for the time range from 1 s to 1000 s, and a linear scale is used for the time range between 0 s and 1 s

mation than simple tables. The results archive (see Table 4) and the web site (<https://sv-comp.sosy-lab.org/2026/results>) include such a plot for each category. As an example, we show the plot for category *C.Overall* in Fig. 4. A total number of 19 verifiers participated in this category. Since SV-COMP 2025, we use different line styles to distinguish regular (i.e., not meta verifiers) active participants (solid line), active meta verifiers (dashed line), and all inactive tools (dotted line). For each participant, the quantile plot shows the overall performance in *C.Overall*. The plot says that the winner of *C.Overall* is **UAUTOMIZER** as its graph ends furthest to the right. The graph for **UAUTOMIZER** also starts slightly left from $x = 0$ because the verifier produced 10 incorrect results and therefore received some negative points. Also other verifiers whose graphs start with a negative cumulative score produced incorrect results. The graphs also illustrate run-time characteristics: The area below a graph is roughly proportional to the accumulated run time. It is easy to see that **GOBLINT** performs best regarding run time. A more detailed discussion of score-based quantile plots, including examples of what insights one can obtain from the plots, is provided in previous competition reports [21, 24].

Validation Track. Validation of verification witnesses was pioneered by SV-COMP in 2015. Shortly after that, verification witnesses become more and more important for various reasons: they do not only justify and help to understand and interpret verification results, but they also serve as exchange object for intermediate results and allow to make use of imprecise verification techniques (e.g., via machine learning). However, a case study on the quality of the results of witness validators [49] published in 2022 revealed a great potential for improvements. To

Table 14: Validation of correctness witnesses in formats 2.0 and 2.1: Overview of the medalists in each meta category; CPU time in hours and rounded to two significant digits

<i>Category</i>			CPU	Solved	Wrong	Wrong
Rank	Validator	Score	Time	Tasks	Confirm.	Refut.
<i>C.ReachSafety</i> (42807 tasks, max. score 64211)						
1	METAVAL	51786	310	38 102	17	0
2	UAUTOMIZER	35695	290	17 806	3	0
3	CPACHECKER	28910	250	31 346	0	16
<i>C.NoOverflows</i> (47221 tasks, max. score 70832)						
1	UAUTOMIZER	63546	380	43 643	1	0
2	CPACHECKER	54068	110	36 689	2	0
3	GOBLINT	39671	2.4	34 788	0	0
<i>C.SoftwareSystems</i> (15708 tasks, max. score 21991)						
1	GOBLINT	9411	25	11 856	0	0
2	UAUTOMIZER	7706	140	12 596	6	0
3	CPACHECKER	4314	20	5 354	0	0
<i>C.ValidationCrafted</i> (18 tasks, max. score 27)						
1	MOPSA	14	0.0080	12	0	0
2	GOBLINT	13	0.0010	11	0	0
3	METAVAL	13	0.068	16	1	0
<i>C.Overall</i> (105754 tasks, max. score 155987)						
1	UAUTOMIZER	89983	810	74 061	11	0
2	CPACHECKER	71251	380	73 398	2	16
3	GOBLINT	65792	36	52 863	0	0

stimulate further advances in verification witnesses and their validators, the study suggested that witness validators should also undergo a periodical comparative evaluation and proposed a scoring schema for witness-validation results. This materializes in the validation track for C programs run by SV-COMP since 2023.

Thanks to the development and adoption of the witness format 2.0 [8], we could eliminate the use of correctness witnesses in format 1.0 from the validation track. Hence, the track has now three subtracks:

1. validation of correctness witnesses in formats 2.0 and 2.1 (see [Table 14](#)),
2. validation of violation witnesses in formats 2.0 and 2.1 (see [Table 15](#)), and
3. validation of violation witnesses in format 1.0 (see [Table 16](#)).

The tables present only the medalists in non-empty meta categories in each subtrack. If some category has less than 3 medalists, it means that less than three validators with active team support reached a positive score. Two non-empty meta categories of the last subtrack, namely *C.NoOverflows* and *C.Overall*, are completely missing in [Table 16](#) as neither of the two validators with active team support participating in these categories reached a positive score. The

Table 15: Validation of violation witnesses in formats 2.0 and 2.1: Overview of the medalists in each meta category; CPU time in hours and rounded to two significant digits

<i>Category</i>						
Rank	Validator	Score	CPU Time	Solved Tasks	Wrong Confirm.	Wrong Refut.
<i>C.ReachSafety</i> (19560 tasks, max. score 30039)						
1	CPACHECKER	16829	50	12 312	4	131
2	WITCH	16204	12	7 605	0	24
3	UAUTOMIZER	8477	69	8 497	1	158
<i>C.MemSafety</i> (655 tasks, max. score 901)						
1	UAUTOMIZER	832	4.8	558	0	0
2	WITCH	791	0.50	653	0	0
3	CPACHECKER	632	1.2	650	0	0
<i>C.NoOverflows</i> (9664 tasks, max. score 12080)						
1	UAUTOMIZER	11196	56	9 625	1	0
2	CPACHECKER	9961	20	9 237	0	0
3	WITCH	9147	9.0	9 112	0	0
<i>C.Termination</i> (1483 tasks, max. score 1483)						
1	WITCH	1359	0.39	1 298	0	0
2	CPACHECKER	1348	12	1 205	0	0
3	UAUTOMIZER	1238	13	1 426	0	0
<i>C.SoftwareSystems</i> (824 tasks, max. score 1288)						
1	WITCH	759	0.62	446	0	0
2	UAUTOMIZER	566	3.1	321	0	0
3	CPACHECKER	524	2.1	455	0	0
<i>C.ValidationCrafted</i> (117 tasks, max. score 176)						
1	WITCH	163	0.041	115	0	0
2	THETA ^{new}	24	0.013	5	0	0
<i>C.Overall</i> (32303 tasks, max. score 44272)						
1	WITCH	33434	22	19 229	0	24
2	CPACHECKER	18482	86	23 916	8	131
3	THETA ^{new}	4665	2.9	539	0	0

column ‘Wrong Confirm.’ gives the number of cases when the respective validator confirmed a witness that was incorrect. The column ‘Wrong Refut.’ shows the number of cases when the respective validator refuted a correct witness. We recall that the correctness or incorrectness of many witnesses used in the validation track is determined by voting and thus the numbers of wrong confirmations and refutations do not have to be completely objective.

The complete results of all validators in all relevant categories of all subtracks are available in the results artifact (see Table 4) and on the SV-COMP web site (<https://sv-comp.sosy-lab.org/2026/results/results-validated/>).

The limited support of some properties and program features by the witness formats (see Table 1), missing medalists in some categories, and the negative scores in the detailed results on the web site: all of these show the need of further research and development in the area of software verification witnesses and their validation.

Table 16: Validation of violation witnesses in format 1.0: Overview of the medalists in each meta category; CPU time in hours and rounded to two significant digits

<i>Category</i>			CPU	Solved	Wrong	Wrong
Rank	Validator	Score	Time	Tasks	Confirm.	Refut.
<i>C.ReachSafety</i> (35069 tasks, max. score 53856)						
1	CONCURRENTW2T	3389	1.7	2 784	26	0
<i>C.MemSafety</i> (9390 tasks, max. score 14085)						
1	CPACHECKER	6653	19	9 181	21	28
<i>C.Concurrency</i> (3799 tasks, max. score 5699)						
1	DARTAGNAN	3299	7.9	1 795	14	0
2	UAUTOMIZER	3199	23	3 268	3	45
3	CPACHECKER	1487	3.6	800	24	11
<i>C.Termination</i> (1685 tasks, max. score 2528)						
1	UAUTOMIZER	1211	12	1 661	0	0
<i>C.SoftwareSystems</i> (7003 tasks, max. score 12255)						
1	CPACHECKER	3893	16	2 962	15	12
2	UAUTOMIZER	1795	45	4 929	3	338

Table 16 indicates diminishing interest of validator developers in supporting the old witness format 1.0. As a consequence, SV-COMP will abandon this format in the future.

6 Conclusion

The 15th edition of the Competition on Software Verification (SV-COMP 2026) compared 74 automatic tools for software verification (including 12 new ones and 29 tools without active team support) and 20 automatic tools for validation of verification witnesses (including 3 new and 6 tools without active team support). The overall numbers of evaluated verifiers and validators were historically the highest (see Fig. 3). The total number of verification tasks with C or Java programs significantly increased to precisely 38 133. On top of that, the competition considered 254 verification tasks in the recently introduced format SV-LIB for software-verification tasks.

The results of the competition show good progress in both verification and witness validation, especially in the adoption and further development of the witness format 2.0. However, even the best verification and validation tools still produce some incorrect results and are far from solving all benchmarks. For example, category *C.ReachSafety* offers 14 375 verification tasks (max. score 23 331), but the best verifier in that category can solve only 8 053 tasks (score 12 866). This motivates further improvements of verifiers. Similar situation holds for validators. For example, out of the 19 560 validation tasks with violation witnesses 2.0 or 2.1 in category *C.ReachSafety* (max. score 30 039), the best validator in that category solved 12 312 tasks (score 16 829). Finally, the competition motivates further extensions of the witness format to support more properties and program features.

SV-COMP received remarkable support from outside its own community. A research group developed the tool ARG-V [122], in order to extend the benchmark set for Java from real software projects found on GitHub. Intel open-sourced their security-critical firmware component Intel® TDX Module and allocated resources to develop and contribute an industrial benchmark set [34]. The TACAS conference acknowledged the importance of benchmarks and comparative evaluations by accepting for publication in this TACAS 2026 proceedings volume the two above works, this report, and 15 competition contributions. Huawei provided travel support to enable researchers to participate in SV-COMP, awarded prizes for a dedicated demo category of challenges in concurrency verification, and supported the infrastructure of the competition.

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

Funding Statement. SV-COMP 2026 was supported by Huawei – Dresden Research Center, Germany. Some participants of this competition were funded in part by the Deutsche Forschungsgemeinschaft (DFG) — 378803395 (ConVeY) and 536040111 (Bridge). Jan Strejček has been supported by the Czech Science Foundation grant GA23-06506S.

Acknowledgments. We thank the organization committee for their help in running the competition and for their work on improving the quality of the verification and validation tasks, the jury for their advice in refining and application of the competition rules and for their work in the paper selection process, and the verification community for contributing their tools to the evaluation and submitting new benchmarks. Furthermore, we thank Schloss Dagstuhl for hosting a seminar that contributed to the development of witness and exchange formats [42] and TACAS for hosting SV-COMP as part of their program at ETAPS.

References

1. m, Zs., Sallai, Gy., Hajdu, .: GAZER-THETA: LLVM-based verifier portfolio with BMC/CEGAR (competition contribution). In: Proc. TACAS (2). pp. 433–437. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_27
2. 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>
3. Aljaafari, F., Shmarov, F., Manino, E., Menezes, R., Cordeiro, L.: EBF 4.2: Black-Box cooperative verification for concurrent programs (competition contribution). In: Proc. TACAS (2). pp. 541–546. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_33

4. Andrianov, P., Friedberger, K., Mandrykin, M.U., Mutilin, V.S., Volkov, A.: CPA-BAM-BNB: Block-abstraction memoization and region-based memory models for predicate abstractions (competition contribution). In: Proc. TACAS (2). pp. 355–359. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_22
5. Andrianov, P., Mutilin, V., Khoroshilov, A.: CPALOCKATOR: Thread-modular approach with projections (competition contribution). In: Proc. TACAS (2). pp. 423–427. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_25
6. Andrianov, P.S.: Analysis of correct synchronization of operating system components. Program. Comput. Softw. **46**, 712–730 (2020). <https://doi.org/10.1134/S0361768820080022>
7. Arceri, V., Negrini, L., Zanatta, G., Bianchi, F., Lisovenko, T., Olivieri, L., Ferrara, P.: JLiSA: The Java frontend of the library for static analysis (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
8. 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
9. Ayaziová, P., Jonáš, M., Mihalkovič, V., Sedláček, J., Strejček, J.: SYMBIOTIC 11: Predicate abstraction joins the party (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
10. Ayaziová, P., Strejček, J.: SYMBIOTIC-WITCH 2: More efficient algorithm and witness refutation (competition contribution). In: Proc. TACAS (2). pp. 523–528. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_30
11. Ayaziová, P., Strejček, J.: WITCH 3: Validation of violation witnesses in the witness format 2.0 (competition contribution). In: Proc. TACAS (3). pp. 341–346. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_18
12. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPACHECKER 3.0: Tutorial and user guide. In: Proc. FM. pp. 543–570. LNCS 14934, Springer (2024). https://doi.org/10.1007/978-3-031-71177-0_30
13. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPACHECKER 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21
14. Bajcezi, L., Ádám, Zs., Micskei, Z.: CONCURRENTWITNESS2TEST: Test-harnessing the power of concurrency (competition contribution). In: Proc. TACAS (3). pp. 330–334. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_16
15. Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Proc. ATVA. pp. 201–207. LNCS 10482, Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_14
16. Barth, M., Beyer, D., Chien, P.C., Jankola, M.: Benchcloud Release 1.5.0. Zenodo (2026). <https://doi.org/10.5281/zenodo.18670174>
17. Barth, M., Dietsch, D., Heizmann, M., Jakobs, M.C.: ULTIMATE PARALIZER: Parallel trace abstraction (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
18. 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

19. Bentele, M., Barth, M., Ebbinghaus, M., Körner, J., Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: *ULTIMATE AUTOMIZER* with a one-dimensional memory model (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
20. Beyer, D.: Competition on software verification (SV-COMP). In: Proc. TACAS. pp. 504–524. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_38
21. 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
22. Beyer, D.: Status report on software verification (Competition summary SV-COMP 2014). In: Proc. TACAS. pp. 373–388. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_25
23. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS. pp. 401–416. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
24. Beyer, D.: Reliable and reproducible competition results with *BENCHEXEC* and witnesses (Report on SV-COMP 2016). In: Proc. TACAS. pp. 887–904. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55
25. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS (2). pp. 331–349. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_20
26. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3). pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
27. Beyer, D.: Advances in automatic software verification: SV-COMP 2020. In: Proc. TACAS (2). pp. 347–367. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_21
28. Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: Proc. TACAS (2). pp. 401–422. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_24
29. 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
30. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
31. 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
32. 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
33. 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>
34. 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, Springer (2026)

35. 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>
36. 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>
37. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). <https://doi.org/10.1145/3477579>
38. 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>
39. 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
40. Beyer, D., Ernst, G., Jonáš, M., Lingsch-Rosenfeld, M.: SV-LIB 1.0: A standard exchange format for software-verification tasks. arXiv/CoRR **2511**(21509) (December 2025). <https://doi.org/10.48550/arXiv.2511.21509>
41. Beyer, D., Friedberger, K.: Violation witnesses and result validation for multi-threaded programs. In: Proc. ISoLA (1). pp. 449–470. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_26
42. Beyer, D., Huisman, M., Strejček, J., Wehrheim, H.: Information exchange in software verification (Dagstuhl Seminar 25172). Dagstuhl Reports **15**(4), 92–111 (2025). <https://doi.org/10.4230/DAGREP.15.4.92>
43. Beyer, D., Jankola, M., Lingsch-Rosenfeld, M.: Transition invariants revisited: Termination witnesses and their validation (2025)
44. Beyer, D., Lingsch-Rosenfeld, M.: CPACHECKER 4.0 as witness validator (competition contribution). In: Proc. TACAS (3). pp. 192–198. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_11
45. 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>
46. Beyer, D., Spiessl, M.: METAVAL: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
47. Beyer, D., Spiessl, M.: The static analyzer FRAMA-C in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 429–434. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_26
48. Beyer, D., Spiessl, M.: LIV: A loop-invariant validation using straight-line programs. In: Proc. ASE. pp. 2074–2077. IEEE (2023). <https://doi.org/10.1109/ASE56229.2023.00214>
49. Beyer, D., Strejček, J.: Case study on verification-witness validators: Where we are and where we go. In: Proc. SAS. pp. 160–174. LNCS 13790, Springer (2022). https://doi.org/10.1007/978-3-031-22308-2_8
50. 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
51. Beyer, D., Strejček, J.: SV-Witnesses – Format 2.1. Zenodo (2025). <https://doi.org/10.5281/zenodo.17277275>
52. Beyer, D., Strejček, J.: Results of the 15th Intl. Competition on Software Verification (SV-COMP 2026). Zenodo (2026). <https://doi.org/10.5281/zenodo.18651757>

53. Beyer, D., Strejček, J.: SV-Benchmarks: Benchmark set for software verification and testing (SV-COMP 2026, Test-Comp 2026). Zenodo (2026). <https://doi.org/10.5281/zenodo.18650775>
54. Beyer, D., Strejček, J.: Verification witnesses from verification tools (SV-COMP 2026). Zenodo (2026). <https://doi.org/10.5281/zenodo.18651735>
55. Beyer, D., Wachowitz, H.: FM-WECK Release 1.6.0. Zenodo (2026). <https://doi.org/10.5281/zenodo.18650812>
56. Biase, P.D., Fischer, B., Torre, S.L., Schrammel, P., Parlato, G.: IEKKE: A SAT-based bounded-round verifier for multi-threaded programs (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
57. Blicha, M., Kofroň, J., Glitta, O.: HORNIX: From LLVM IR to constrained Horn clauses and back (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
58. Brablec, T., Dacík, T., Vojnar, T.: SEAL: Symbolic execution with separation logic (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
59. Brain, M., Joshi, S., Kröning, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Proc. SAS. pp. 145–161. LNCS 9291, Springer (2015). https://doi.org/10.1007/978-3-662-48288-9_9
60. Bu, L., Xie, Z., Lyu, L., Li, Y., Guo, X., Zhao, J., Li, X.: BRICK: Path enumeration-based bounded reachability checking of C programs (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_22
61. Calcagno, C., Distefano, D., O’Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. *J. ACM* **58**(6), 26:1–26:66 (2011). <https://doi.org/10.1145/2049697.2049700>
62. Chalupa, M., Henzinger, T.: BUBAAK: Runtime monitoring of program verifiers (competition contribution). In: Proc. TACAS (2). pp. 535–540. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_32
63. Chalupa, M., Richter, C.: BUBAAK-SPLIT: Split what you cannot verify (competition contribution). In: Proc. TACAS (3). pp. 353–358. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_20
64. Chalupa, M., Richter, C.: BUBAAK: Dynamic cooperative verification (competition contribution). In: Proc. TACAS (3). pp. 212–216. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_14
65. 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
66. Chaudhary, E., Joshi, S.: PINAKA: Symbolic execution meets incremental solving (competition contribution). In: Proc. TACAS (3). pp. 234–238. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_20
67. Cheng, Z., Wu, T., Schrammel, P., Tihanyi, N., de Lima Filho, E.B., Cordeiro, L.C.: JCWIT: A correctness-witness validator for Java programs based on bounded model checking. In: Proc. ISSTA. pp. 1831–1835. ACM (2024). <https://doi.org/10.1145/3650212.3685303>
68. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier (competition contribution). In: Proc. TACAS (3). pp. 365–370. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_22
69. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proc. TACAS. pp. 168–176. LNCS 2988, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15

70. Cordeiro, L.C., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: Proc. ICSE. pp. 331–340. ACM (2011). <https://doi.org/10.1145/1985793.1985839>
71. Cordeiro, L.C., Kesseli, P., Kröning, D., Schrammel, P., Trtík, M.: JBMC: A bounded model checking tool for verifying Java bytecode. In: Proc. CAV. pp. 183–190. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_10
72. Cordeiro, L.C., Kröning, D., Schrammel, P.: JBMC: Bounded model checking for Java bytecode (competition contribution). In: Proc. TACAS (3). pp. 219–223. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_17
73. Cordeiro, L.C., Morse, J., Nicole, D., Fischer, B.: Context-bounded model checking with ESBMC 1.17 (competition contribution). In: Proc. TACAS. pp. 534–537. LNCS 7214, Springer (2012). https://doi.org/10.1007/978-3-642-28756-5_42
74. Coto, A., Inverso, O., Sales, E., Tuosto, E.: A prototype for data race detection in CSEQ 3 (competition contribution). In: Proc. TACAS (2). pp. 413–417. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_23
75. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: FRAMA-C. In: Proc. SEFM. pp. 233–247. Springer (2012). https://doi.org/10.1007/978-3-642-33826-7_16
76. Dacík, T., Vojnar, T.: Racerf: Lightweight static data race detection for C code (experience paper). In: Proc. ECOOP. pp. 37:1–37:19. LIPIcs 333, Schloss Dagstuhl (2025). <https://doi.org/10.4230/LIPIcs.ECOOP.2025.37>
77. Dacík, T., Vojnar, T.: RACERF: Data race detection with Frama-C (competition contribution). In: Proc. TACAS (3). pp. 248–253. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_20
78. Darke, P., Agrawal, S., Venkatesh, R.: VERIABS: A tool for scalable verification by abstraction (competition contribution). In: Proc. TACAS (2). pp. 458–462. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_32
79. Darke, P., Chimdyalwar, B., Agrawal, S., Venkatesh, R., Chakraborty, S., Kumar, S.: VERIABSL: Scalable verification by abstraction and strategy prediction (competition contribution). In: Proc. TACAS (2). pp. 588–593. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_41
80. Denis, X., Siegel, S.F.: VerifyThis 2023: An international program verification competition. In: TOOLympics Challenge 2023 - Updates, Results, Successes of the Formal-Methods Competitions, TOOLympics 2023, Paris, France. pp. 147–159. LNCS 14550, Springer (2023). https://doi.org/10.1007/978-3-031-67695-6_5
81. Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: ULTIMATE TAIPAN 2023 (competition contribution). In: Proc. TACAS (2). pp. 582–587. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_40
82. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: Proc. VSTTE. pp. 56–72. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_5
83. Erhard, J., Bentele, M., Heizmann, M., Klumpp, D., Saan, S., Schüssele, F., Schwarz, M., Seidl, H., Tilscher, S., Vojdani, V.: Correctness witnesses for concurrent programs: Bridging the semantic divide with ghosts. In: Proc. VMCAI, Part I. pp. 74–100. LNCS 15529, Springer (2025). https://doi.org/10.1007/978-3-031-82700-6_4
84. Ermis, E., Hoenicke, J., Podelski, A.: Splitting via interpolants. In: Proc. VMCAI. pp. 186–201. LNCS 7148, Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_13

85. Ernst, G.: A complete approach to loop verification with invariants and summaries. Tech. Rep. arXiv:2010.05812v2, arXiv (January 2020). <https://doi.org/10.48550/arXiv.2010.05812>
86. Ernst, G.: KORN: Horn clause based verification of C programs (competition contribution). In: Proc. TACAS (2). pp. 559–564. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_36
87. Farzan, A., Klumpp, D., Podelski, A.: Sound sequentialization for concurrent program verification. In: Proc. PLDI. pp. 506–521. ACM (2022). <https://doi.org/10.1145/3519939.3523727>
88. 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>
89. Gavrilenko, N., Ponce de León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: Relation analysis for compact SMT encodings. In: Proc. CAV. pp. 355–365. LNCS 11561, Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_19
90. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Proc. CADE. pp. 105–108. LNCS 9195, Springer (2015). https://doi.org/10.1007/978-3-319-21401-6_6
91. Greitschus, M., Dietsch, D., Podelski, A.: Loop invariants from counterexamples. In: Proc. SAS. pp. 128–147. LNCS 10422, Springer (2017). https://doi.org/10.1007/978-3-319-66706-5_7
92. Hajdu, Á., Micskei, Z.: Efficient strategies for CEGAR-based model checking. J. Autom. Reasoning **64**(6), 1051–1091 (2020). <https://doi.org/10.1007/s10817-019-09535-x>
93. He, F., Sun, Z., Fan, H.: DEAGLE: An SMT-based verifier for multi-threaded programs (competition contribution). In: Proc. TACAS (2). pp. 424–428. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_25
94. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Proc. CAV. pp. 36–52. LNCS 8044, Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2
95. Heizmann, M., Klumpp, D., Lingsch-Rosenfeld, M., Schüssele, F.: Correctness witnesses with function contracts. arXiv/CoRR **2501**(12313) (January 2025). <https://doi.org/10.48550/arXiv.2501.12313>
96. Holík, L., Kotoun, M., Peringer, P., Šoková, V., Trtík, M., Vojnar, T.: PREDATOR shape analysis tool suite. In: Hardware and Software: Verification and Testing. pp. 202–209. LNCS 10028, Springer (2016). <https://doi.org/10.1007/978-3-319-49052-6>
97. Holter, K., Ayaziová, P., Saan, S., Strejček, J., Vojdani, V.: GOBLITCH: Combining abstract interpretation with symbolic execution via witnesses (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
98. Howar, F., Jasper, M., Mues, M., Schmidt, D.A., Steffen, B.: The RERS challenge: Towards controllable and scalable benchmark synthesis. Int. J. Softw. Tools Technol. Transf. **23**(6), 917–930 (2021). <https://doi.org/10.1007/s10009-021-00617-z>
99. Howar, F., Mues, M.: GWIT (competition contribution). In: Proc. TACAS (2). pp. 446–450. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_29
100. Hussein, S., Yan, Q., McCamant, S., Sharma, V., Whalen, M.: JAVA RANGER: Supporting string and array operations (competition contribution). In: Proc. TACAS (2). pp. 553–558. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_35

101. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: LAZY-CSEQ: A lazy sequentialization tool for C (competition contribution). In: Proc. TACAS. pp. 398–401. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_29
102. Inverso, O., Tomasco, E., Fischer, B., Torre, S.L., Parlato, G.: Bounded verification of multi-threaded programs via lazy sequentialization. *ACM Trans. Program. Lang. Syst.* **44**(1), 1:1–1:50 (2022). <https://doi.org/10.1145/3478536>
103. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: Proc. PPOPP. pp. 202–216. ACM (2020). <https://doi.org/10.1145/3332466.3374529>
104. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Proc. VSTTE. pp. 1–18. LNCS 12031, Springer (2019). https://doi.org/10.1007/978-3-030-41600-3_1
105. Kahsai, T., Rümmer, P., Sanchez, H., Schäfer, M.: JAYHORN: A framework for verifying Java programs. In: Proc. CAV. pp. 352–358. LNCS 9779, Springer (2016). https://doi.org/10.1007/978-3-319-41528-4_19
106. Kettl, M., Lemberger, T.: The static analyzer INFER in SV-COMP (competition contribution). In: Proc. TACAS (2). pp. 451–456. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_30
107. Klumpp, D., Dietsch, D., Heizmann, M., Schüssele, F., Ebbinghaus, M., Farzan, A., Podelski, A.: ULTIMATE GEMCUTTER and the axes of generalization (competition contribution). In: Proc. TACAS (2). pp. 479–483. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_35
108. Kröning, D., Tautschnig, M.: CBMC: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_26
109. Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_17
110. Leeson, W., Dwyer, M.: GRAVES-CPA: A graph-attention verifier selector (competition contribution). In: Proc. TACAS (2). pp. 440–445. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28
111. Lemberger, T., Wachowitz, H.: NACPA: Native checking with parallel-portfolio analyses (competition contribution). In: Proc. TACAS (3). pp. 236–241. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_18
112. de Leon, H.P., Furbach, F., Heljanko, K., Meyer, R.: DARTAGNAN: Bounded model checking for weak memory models (competition contribution). In: Proc. TACAS (2). pp. 378–382. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_24
113. Lin, Y., Chen, Z., Wang, J.: AISE v2.0: Combining loop transformations (competition contribution). In: Proc. TACAS (3). pp. 199–204. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_12
114. Lommen, N., Giesl, J.: APROVE (KoAT+LoAT) (competition contribution). In: Proc. TACAS (3). pp. 205–211. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_13
115. Loose, N., Mächtle, F., Sieck, F., Eisenbarth, T.: SWAT: Modular dynamic symbolic execution for Java applications using dynamic instrumentation (competition contribution). In: Proc. TACAS (3). pp. 399–405. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_28
116. Loose, N., Sieck, F., Mächtle, F., Eisenbarth, T.: SWAT: Improvements to the symbolic executor (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)

117. Luckow, K.S., Dimjasevic, M., Giannakopoulou, D., Howar, F., Isberner, M., Kahsai, T., Rakamaric, Z., Raman, V.: JDART: A dynamic symbolic analysis framework. In: Proc. TACAS. pp. 442–459. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_26
118. Malík, V., Schrammel, P., Vojnar, T., Nečas, F.: 2LS: Arrays and loop unwinding (competition contribution). In: Proc. TACAS (2). pp. 529–534. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_31
119. McGowan, C., Richards, M., Sui, Y.: SVF-SVC: Software verification using SVF (competition contribution). In: Proc. TACAS (3). pp. 254–259. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_21
120. Metta, R., Karmarkar, H., Madhukar, K., Venkatesh, R., Chakraborty, S.: PROTON: Probes for non-termination and termination (competition contribution). In: Proc. TACAS (3). pp. 393–398. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_27
121. Milanese, M., Monat, R., Oudjaout, A., Miné, A.: MOPSA-C: Towards incorrectness and termination verdicts (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
122. Moloney, C., Dyer, R., Sherman, E.: Demonstrating ARG-V’s generation of realistic Java benchmarks for SV-COMP. In: Proc. TACAS. Springer (2026)
123. Monat, R., Milanese, M., Parolini, F., Boillot, J., Oudjaout, A., Miné, A.: MOPSA-C: Improved verification for C programs, simple validation of correctness witnesses (competition contribution). In: Proc. TACAS (3). pp. 387–392. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_26
124. Mondok, M., Bajczi, L., Szekeres, D., Molnár, V.: EMERGENTHETA: Variations on symbolic transition systems (competition contribution). In: Proc. TACAS (3). pp. 217–222. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_15
125. Mondok, M., Telbisz, C., Bajczi, L., Kovács, D., Dobos-Kovács, M., Molnár, V.: EMERGENTHETA: Experimental analyses within the theta framework (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
126. Moussaoui-Remil, N., Urban, C.: REFUNCTION: Conditional termination by abstract interpretation of numerical C programs (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
127. Mues, M., Howar, F.: JDART: Portfolio solving, breadth-first search and SMT-Lib strings (competition contribution). In: Proc. TACAS (2). pp. 448–452. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_30
128. Mues, M., Howar, F.: GDART (competition contribution). In: Proc. TACAS (2). pp. 435–439. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_27
129. Mukhopadhyay, D., Metta, R., Karmarkar, H., Madhukar, K.: PROTON 2.1: Synthesizing ranking functions via fine-tuned locally hosted LLM (competition contribution). In: Proc. TACAS (3). pp. 242–247. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_19
130. Mächtle, F., Serr, J.N., Loose, N., Eisenbarth, T.: DASA: Fully gradient-based program analysis (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
131. Noller, Y., Păsăreanu, C.S., Le, X.B.D., Visser, W., Fromherz, A.: Symbolic PATHFINDER for SV-COMP (competition contribution). In: Proc. TACAS (3). pp. 239–243. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_21

132. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks (competition contribution). In: Proc. TACAS. pp. 458–460. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_44
133. Peringer, P., Šoková, V., Vojnar, T.: PREDATORHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution). In: Proc. TACAS (2). pp. 408–412. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_30
134. Ponce-De-Leon, H., Haas, T., Meyer, R.: DARTAGNAN: SMT-based violation witness validation (competition contribution). In: Proc. TACAS (2). pp. 418–423. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_24
135. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Practical static race detection for C. ACM Trans. Program. Lang. Syst. **33**(1) (January 2011). <https://doi.org/10.1145/1889997.1890000>
136. Păsăreanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic PATHFINDER: Integrating symbolic execution with model checking for Java bytecode analysis. Autom. Software Eng. **20**(3), 391–425 (2013). <https://doi.org/10.1007/s10515-013-0122-2>
137. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). <https://doi.org/10.1007/s10515-020-00270-x>
138. Richter, C., Wehrheim, H.: PESCO: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
139. Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: GOBLINT VALIDATOR: Correctness witness validation by abstract interpretation (competition contribution). In: Proc. TACAS (3). pp. 335–340. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_17
140. Saan, S., Kocal, A.R., Petter, M., Holter, K., Erhard, J., Schwarz, M., Vojdani, V., Seidl, H.: GOBLINT: A portfolio for mixed flow-sensitive abstract interpretation (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
141. Scott, R., Dockins, R., Ravitch, T., Tomb, A.: CRUX: Symbolic execution meets SMT-based verification (competition contribution). Zenodo (February 2022). <https://doi.org/10.5281/zenodo.6147218>
142. Shamakhi, A., Hojjat, H., Rümmer, P.: Towards string support in JAYHORN (competition contribution). In: Proc. TACAS (2). pp. 443–447. LNCS 12652, Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_29
143. Sharma, V., Hussein, S., Whalen, M.W., McCamant, S.A., Visser, W.: JAVA RANGER: Statically summarizing regions for efficient symbolic execution of Java. In: Proc. ESEC/FSE. pp. 123–134. ACM (2020). <https://doi.org/10.1145/3368089.3409734>
144. Su, J., Yang, Z., Xing, H., Yang, J., Tian, C., Duan, Z.: PICHECKER: A POR and interpolation-based verifier for concurrent programs (competition contribution). In: Proc. TACAS (2). pp. 571–576. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_38
145. Telbisz, C., Bajczi, L., Szekeres, D., Vörös, A.: THETA: Various approaches for concurrent program verification (competition contribution). In: Proc. TACAS (3). pp. 260–265. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_22
146. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: THETA: A framework for abstraction refinement-based model checking. In: Proc. FMCAD. pp. 176–179 (2017). <https://doi.org/10.23919/FMCAD.2017.8102257>

147. Visser, W., Geldenhuys, J.: COASTAL: Combining concolic and fuzzing for Java (competition contribution). In: Proc. TACAS (2). pp. 373–377. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_23
148. Vojdani, V., Apinis, K., Rötov, V., Seidl, H., Vene, V., Vogler, R.: Static race detection for device drivers: The Goblint approach. In: Proc. ASE. pp. 391–402. ACM (2016). <https://doi.org/10.1145/2970276.2970337>
149. Volkov, A.R., Mandrykin, M.U.: Predicate abstractions memory modeling method with separation into disjoint regions. Proceedings of the Institute for System Programming (ISPRAS) **29**, 203–216 (2017). [https://doi.org/10.15514/ISPRAS-2017-29\(4\)-13](https://doi.org/10.15514/ISPRAS-2017-29(4)-13)
150. Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution (competition contribution). In: Proc. TACAS (3). pp. 347–352. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_19
151. Wendler, P., Beyer, D.: sosy-lab/benchexec: Release 3.34. Zenodo (2026). <https://doi.org/10.5281/zenodo.18455156>
152. Wu, T., Li, X., Manino, E., Menezes, R., Gadelha, M., Xiong, S., Tihanyi, N., Petoumenos, P., Cordeiro, L.: ESBMC v7.7: Efficient concurrent software verification with scheduling, incremental SMT and partial order reduction (competition contribution). In: Proc. TACAS (3). pp. 223–228. LNCS 15698, Springer (2025). https://doi.org/10.1007/978-3-031-90660-2_16
153. Wu, T., Schrammel, P., Cordeiro, L.: WIT4JAVA: A violation-witness validator for Java verifiers (competition contribution). In: Proc. TACAS (2). pp. 484–489. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_36
154. Ādám, Zs., Ayaziová, P., Bajczi, L., Beyer, D., Jankola, M., Lingsch-Rosenfeld, M., Strejček, J.: Non-termination witnesses and their validation. In: Proc. ASE 2025. IEEE (2025)
155. Štěpková, A., Jonáš, M., Strejček, J.: RE3VER: Reverse and verify (competition contribution). In: Proc. TACAS (2). LNCS 16506, Springer (2026)
156. J. Švejda, Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C: NITWIT. In: Proc. TACAS. pp. 40–57. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_3

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

