

HarnessForge: Automated Extraction of Verification Tasks from Industry-Scale Software Projects

Dirk Beyer
LMU Munich
Germany

Po-Chun Chien
LMU Munich
Germany

Bo-Yuan Huang
Intel INT31
USA

Nian-Ze Lee*
National Taiwan University
Taiwan

Thomas Lemberger
LMU Munich
Germany

Abstract

We present HARNESFORGE, a command-line tool to streamline the extraction of verification tasks from industry-scale software projects written in C. Industry-scale code consists of multiple source and header files with various build processes, complicating the creation of verification tasks and hindering the applicability of off-the-shelf software verifiers. HARNESFORGE handles this complexity for verification engineers and tools, allowing harnesses to be structured independently from the code under verification. It automatically derives build commands, assembles relevant source files, and performs static program slicing to remove irrelevant components. To demonstrate its applicability, we use HARNESFORGE to create a total of 949 verification tasks from three projects: AWS C Common, GNU Coreutils, and Intel TDX Module. All created tasks were used in SV-COMP 2026. A demo video is available at youtu.be/wHPEfQ3NBfQ.

CCS Concepts

• **General and reference** → **Verification**; • **Software and its engineering** → **Software verification**; **Preprocessors**; *Software testing and debugging*; Automated static analysis.

Keywords

Industry-scale software, Harnesses, Verification tasks, Program slicing, Software verification, Software testing

ACM Reference Format:

Dirk Beyer, Po-Chun Chien, Bo-Yuan Huang, Nian-Ze Lee, and Thomas Lemberger. 2026. HarnessForge: Automated Extraction of Verification Tasks from Industry-Scale Software Projects. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26)*, July 05–09, 2026, Montreal, QC, Canada. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3803437.3806420>

*Also with LMU Munich.



This work is licensed under a Creative Commons Attribution 4.0 International License. *FSE Companion '26, Montreal, QC, Canada*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2636-1/2026/07
<https://doi.org/10.1145/3803437.3806420>

1 Introduction

The precise analysis techniques offered by formal software verifiers [4, 9, 17, 27, 28, 37] are an indispensable complement to testing. They explore unlikely error paths that may be difficult to trigger and provide more abstract counterexamples that aid in understanding software bugs. In testing, test harnesses (1) set up the execution context and test inputs, (2) introduce mocks for external dependencies, (3) execute the component under test, and (4) assert that the behavior observed during execution matches the specification. Such test harnesses are usually separated from application code to maintain modularity and reusability. Formal verifiers require similar harnesses that (1) model the relevant program context and symbolic inputs, (2) introduce mocks for external dependencies, (3) impose *preconditions* on symbolic inputs and data states, (4) call the component under verification, and (5) assert that *postconditions* hold after calling the component with the given program context.

Unlike testing frameworks, which typically integrate with a project's existing build system, most academic formal verifiers expect a single source file that contains the complete verification task. Up to now, users have to manually assemble such single-file verification tasks or generate them through custom scripts. This process is particularly time-consuming and error-prone for industry-scale C projects, which span multiple files organized across header and source directories, and rely on complex build procedures that define macros and resolve dependencies via code inclusion or linking. One seemingly straightforward solution for creating a verification task is to just include all project files. But because files that do not belong together may conflict, this still requires hours of manual trial-and-error for industry-scale projects.

Contributions. HARNESFORGE supports verification engineers in assembling such single-file verification tasks: Configurations of verification tasks are organized independently from the code under verification. HARNESFORGE then automatically assembles a verification task and removes definitions that are irrelevant to the task, while preserving the original code structure.

We used HARNESFORGE to create 949 distinct verification tasks from three industry-scale projects: the **AWS C Common** library, the **GNU Coreutils** project, and the **Intel TDX** Module firmware. These tasks were used in the Competitions on Software Verification (SV-COMP 2026) [11] and Testing (Test-Comp 2026) [5]. On average, HARNESFORGE reduced the lines of code per task by 60 %.

HARNESFORGE is analyzer-agnostic. It has been used [6] to create verification tasks tailored to **CBMC** [17], **ESBMC** [25], **KLEE** [15], and tools that adhere to the conventions of SV-COMP and Test-Comp.

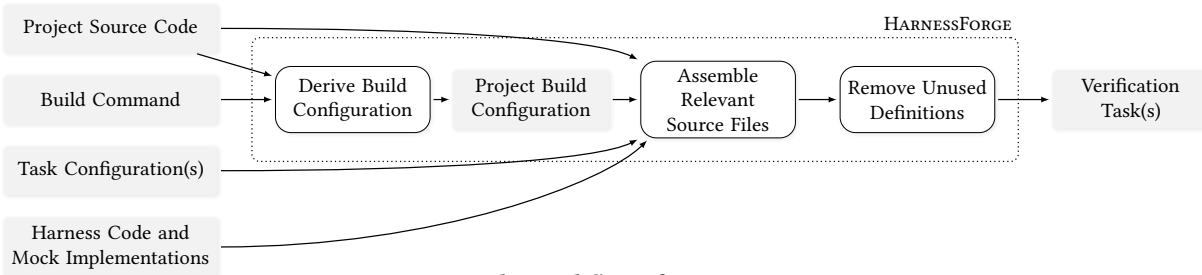


Figure 1: The workflow of HARNESSFORGE

Related Work. ARG-V [34] and PAClab [13] are infrastructures aimed at the creation of verification tasks for benchmarking software verifiers. They mine verification tasks from source-code repositories by identifying files of interest based on user-specified selection criteria, and apply code transformations to make them compilable and suitable for verification. In contrast, HARNESSFORGE does not modify the code under verification. Instead, it assembles concise single-file verification tasks from complex build procedures using user-provided pre- and postcondition pairs, thereby enhancing the interpretability of verification results.

One Line Scan [1] hooks into build systems to run verifiers on a project. Each verifier is executed by a custom backend script that defines how to run the verifier on the detected source files. One Line Scan itself does not differentiate between methods under verification but runs backend scripts on all project source files, making it difficult to use different verification harnesses for different methods. In contrast, HARNESSFORGE uses BEAR [36] to hook into build systems and creates verification tasks for specific methods under verification with provided verification harnesses. Compared to existing source-code slicers [22, 37], HARNESSFORGE does not modify the code of relevant definitions, but only removes type, variable, and function definitions that are irrelevant to the verification task.

2 From Harnesses to Verification Tasks

Overview. Figure 1 shows the workflow of HARNESSFORGE. HARNESSFORGE is a command-line tool that receives four inputs: (1) the project source code, (2) the command that is used to build the project, (3) the harness code that defines preconditions, postconditions, and mock implementations, and (4) one or more *task configuration* files that describe the verification tasks to forge.

First, HARNESSFORGE runs the build command and records all compiler calls in a project build configuration for user adjustments. Next, HARNESSFORGE assembles a self-contained, single C file for each task listed in the task configuration file(s). It uses the provided inputs and the derived build configuration for this, and the assembled C file includes all necessary code dependencies. Finally, HARNESSFORGE statically analyzes the assembled C file and removes definitions for data types and functions that are not reachable from the main function. The result is a verification task that can be consumed by off-the-shelf analyzers.

Deriving the Project Build Configuration. HARNESSFORGE uses the tool BEAR [36] to record a *compilation database*, a JSON file capturing compiler invocations during the build process, while running the provided build command (e.g., `make`). HARNESSFORGE

```
source_dirs:
  - src/
  - include/auto_gen/
  - formal/src/ # added by engineer
includes:
  - include/
  - src/
  - src/common/
  - ...
  - formal/include/ # added by engineer
defs:
  - FAULT_SAFE_MAGIC_INDICATOR: 0xFF0F0F0F0F0F0FFF
  - TDX_MODULE_MAJOR_VER: 1
  - ...
  - TDXFV_NO_ASM # added by engineer
  - TDXFV_ASSIGN_EQ_PTR # added by engineer
override_dir:
  - formal/tdx_override/ # added by engineer
```

Figure 2: Build configuration created by HARNESSFORGE for Intel TDX Module [21] (commented entries manually added to the initial build configuration created by HARNESSFORGE)

extracts all read source files, macro definitions (defined with flag `-D`), and include directories (defined with flag `-I`), and stores them in a YAML file that is easy to read and adjust for verification engineers.

Figure 2 shows an excerpt of the project build configuration derived by HARNESSFORGE after we apply it to Intel TDX Module. Most entries are automatically generated, but verification engineers (in this case, our industrial collaborator at Intel) added entries to make HARNESSFORGE consider sources situated in `formal/src` and `formal/include`. These directories contain code for verification harnesses. Verification engineers also added additional macro definitions to configure the behavior of their verification harnesses. The macro `TDXFV_NO_ASM` is used to toggle the modeling of inline assembly through C code, and the macro `TDXFV_ASSIGN_EQ_PTR` is used to change the method of pointer initialization. This shows that the project build configuration not only records how to build the original project, but that it can also be used to configure the task generation for different analysis scenarios.

Mocking Functionality. To achieve efficient verification, it is often necessary to mock external code dependencies (e.g., to model the behavior of an externally declared method whose source code is not available) or internal code dependencies (e.g., to simplify complex but unimportant methods or to model inline assembly that is not understood by most verifiers).

External dependencies can be modeled by adding code that defines the intended functionality for the external variables and methods. Mocking internal dependencies is not possible in the same way because definitions already exist in the project’s source code.

```

name: tdh_sys_config_config.yml
description: "reach-safety tasks for tdh_sys_config"
tasks:
  - name: tdh_sys_config__expected__safety_requirement
    before_target:
      - filename: formal/src/initialization.c
        method: init_tdh_sys_config
      - filename: formal/harness/tdh_sys_config_harness.c
        method: tdh_sys_config__expected__precond
    target:
      filename: formal/harness/tdh_sys_config_harness.c
      method: tdh_sys_config__call
    after_target:
      - filename: formal/harness/tdh_sys_config_harness.c
        method: tdh_sys_config__expected__postcond
      - filename: formal/harness/tdh_sys_config_harness.c
        method: tdh_sys_config__free
      - filename: formal/src/initialization.c
        method: close_tdh_sys_config_with_check
    properties:
      - property_file: prps/unreach-call.prp
        expected_verdict: true

```

Figure 3: A task configuration file for Intel-TDX interface method `tdh_sys_config` (called through `tdh_sys_config__call`)

To mock internal dependencies, HARNESFORGE provides the parameter `override_dir`, which points to a directory whose files are used to override the original source files when the verification task is assembled. For example, Fig. 2 points `override_dir` to `formal/tdx_override/`, which may tell HARNESFORGE to override an original source file `src/common/helpers/helpers.c` with the specified file `formal/tdx_override/src/common/helpers/helpers.c` (provided that file exists). Users are responsible for ensuring that overrides do not miss any definitions of the overridden source file.

Description of Verification Tasks. Verification engineers can describe verification tasks in a YAML file that lists the methods to be invoked in each verification task and invoke HARNESFORGE to assemble verification tasks. Figure 3 shows an example task configuration file. Each task configuration file has fields `name` and `description` for documentation purposes. It then defines a list of tasks. Each task has fields `name` and three categories of method calls: The field `before_target` lists methods that should be called before the actual method under verification. These methods usually contain set-up code like the initialization of global variables and the assumptions of preconditions. The field `target` references the method under verification, usually through a wrapper method that calls the original method with the parameters initialized in the `before_target` step. The field `after_target` lists methods that should be called after the `target` method under verification. These methods usually assert postconditions for verifiers to prove. All methods in the three categories are referenced by their source file and method name, and will be called in the order they are listed in the task configuration file. They must receive no arguments, and their return values are ignored. HARNESFORGE does not put any other restrictions on the methods called in each of the three steps, to be as generic as possible.

Besides structuring the verification task, users can document the expected verdict of the verification task with respect to different property files (i.e., specifications) under the field `properties`.

Assembly of Single-File Verification Tasks. To assemble a single-file verification task, HARNESFORGE sets up a temporary copy of

```

int main() {
  // before_target
  init_tdh_sys_config();
  tdh_sys_config__expected__precond();
  // target
  tdh_sys_config__call();
  // after_target
  tdh_sys_config__expected__postcond();
  tdh_sys_config__free();
  close_tdh_sys_config_with_check();
  return 0;
}

```

Figure 4: Function `main` created by HARNESFORGE from Fig. 3

the project source code, adds the harness files, and applies any existing overrides. Next, for each of the methods defined in the task configuration file (in `before_target`, `target`, and `after_target`), HARNESFORGE computes the method's call graph and collects the used types and global variables in every reachable method definition. It then includes every source file that contains at least one used definition. Last, HARNESFORGE adds a `main` method that calls the methods listed in `before_target`, `target`, and `after_target` in sequence (see Fig. 4). The original `main` method is renamed if it is among the reachable functions.

Slicing of Unused Definitions. The single file assembled by HARNESFORGE may still contain unused definitions. These may contain code that has dependencies to definitions in source files that are not required by the method under verification. If they remain in the verification task, they not only bloat the verification task with unreachable code, but can make the task fail to compile because their dependencies are missing. For example, assume a method that is never called by the code under verification is defined in a required source file, and this method uses a type that is defined in another source file that is not required for the verification task. If the unreachable method is included in the verification task, it will not compile because the type definition is missing. In addition, unused definitions unnecessarily increase the risk of including code constructs that are not supported by the employed verifiers.

To remove irrelevant definitions, HARNESFORGE first preprocesses the assembled file and uses the previously collected reachable definitions to remove all other definitions (and declarations) from the preprocessed source. These are all unused and thus not relevant.

3 Evaluation

We apply HARNESFORGE to three real-world projects of different domains: the [AWS C Common](#) library, the [GNU Coreutils](#) project, and the [Intel TDX](#) Module firmware.

AWS C Common. The C library [AWS C Common](#) contains the interface code (cross-platform primitives, configuration, data structures, error handling) for the AWS SDK for C developers to access cloud resources. The codebase already contains verification harnesses for creating single-file verification tasks for [CBMC](#) [16]. To validate the functionality of HARNESFORGE, we reproduced 20 of these existing verification tasks with method definitions that adhere to the SV-COMP rules. On average, each task consists of 4 original source files. The static slicing of HARNESFORGE reduced the task size from an average of 4 214 lines of code (2 991 – 4 400) down to an average of 260 lines of code (206 – 317), a reduction

by 94% (91% – 95%). We contributed 12 of these sliced tasks to SV-COMP 2026 [18].

Coreutils. The GNU Coreutils project contains command-line utilities (such as `ls`, `mkdir`, and `echo`) commonly used on UNIX-like systems. Previous work [14] created verification tasks from Coreutils for symbolic execution to find potential memory errors. We complement these tasks with 93 new tasks that focus on memory safety (48 tasks) and functional properties (45 tasks) of selected methods across the Coreutils tools `factors`, `getlimits`, `reldir`, and `seq`. The 45 tasks for functional properties share their verification harness with corresponding memory-safety tasks. The verification harnesses for these methods are manually created. The initialization of non-deterministic values is performed with SV-COMP’s dedicated methods `<type> __VERIFIER_nondet_<type>()`.

On average, each task consists of 3 original source files. Static slicing reduced the size of tasks from an average of 4 686 lines of code (3 961 – 7 320) down to an average of 1 159 lines of code (600 – 3 292), a reduction by 75% (55% – 86%). We contributed the 93 tasks to SV-COMP 2026 [19].

Intel TDX Module. TDX Module is a key firmware component in Intel Trust Domain Extensions, Intel’s latest technology for hardware-assisted confidential computing. We created 418 verification tasks for functional correctness of 22 different interface functions in two different variants for non-deterministic initialization (resulting in a total of 836 tasks): The first variant initializes all fields of complex data structures (e.g., arrays and structs) by setting all elements explicitly, one by one, to non-deterministic values. The second variant uses the SV-COMP-provided function `__VERIFIER_nondet_memory()` to non-deterministically initialize complex data structures. In this case the verifier can decide how to handle the initialization of non-deterministic memory.

On average, each task consists of 15 original source files (9 – 28). Static slicing reduced the size of tasks from an average of 15 624 lines of code (12 497 – 26 574) down to an average of 6 247 lines of code (2 952 – 16 971), a reduction by 60% (36% – 77%). These tasks are used in a case study for firmware verification [6] and we contributed them to SV-COMP 2026 [20].

Effect of Slicing. We investigated the effect of HARNESSFORGE’s static slicing on six different C verifiers: CBMC [17] version 6.6.0 [31], CPAchecker [3, 4] version 4.2.2 [12], and ESBMC [24, 25, 38], MOPSA [30, 35], SYMBIOTIC [29, 37], and UAUTOMIZER [26, 27] in their respective SV-COMP 2026 version [2, 23, 32, 33].

We ran experiments on Ubuntu 24.04 (64 bit) machines with an Intel Xeon E3-1230 v5 CPU (3.4 GHz, 8 cores) and 33 GB of RAM. Each verification run was limited to 4 cores, 15 min of CPU time, and 15 GB of memory. We used BENCHEXEC [10] and BENCHCLOUD [8] for reliable and scalable benchmarking. All 949 verification tasks generated from the three above projects were evaluated: AWS C Common (20), Coreutils (93), and Intel TDX Module (836).

On memory consumption, slicing has no noticeable effect across all considered verifiers. On verification run time, slicing provides a small reduction of verification run time on most tasks. The run time of CPAchecker benefits the most from slicing: Here, the average CPU time for tasks that can be solved both before and after slicing (all tasks from the Intel TDX Module) decreases from 334 s to 307 s.

Table 1: Effect of slicing on the verification effectiveness

	AWS C Common 20 tasks		Coreutils 93 tasks		Intel TDX 836 tasks	
	Before	After	Before	After	Before	After
CBMC	0	2	0	0	93	93
CPACHECKER	0	20	0	27	143	146
ESBMC	17	17	46	46	35	35
MOPSA	20	20	15	17	0	0
SYMBIOTIC	20	20	40	40	0	0
UAUTOMIZER	2	20	0	16	0	0
Sum	59	99	101	146	271	274

On 5 tasks of Intel TDX Module, slicing has a stronger effect on the run time of CPAchecker: On these tasks, CPAchecker can not solve the task within 900 s before slicing, but is able to solve the tasks after slicing with an average CPU time of 593 s. This shows that static slicing of HARNESSFORGE can yield a significant improvement in verification run time, in some cases.

Regarding effectiveness, Table 1 shows the number of correct verification results that each verifier produced for the verification tasks created with HARNESSFORGE before and after slicing. We divide the results by the three considered projects. Verifiers produce 90 additional correct results. They manage to solve 85 additional tasks because parsing errors are resolved; this can be attributed to the irrelevant use of missing definitions being removed and code constructs being eliminated that are unsupported by the verifier. The remaining 5 improvements are the CPAchecker speed-ups mentioned above. But there are also two correct results before slicing that are not present after slicing: CPAchecker can solve two tasks from the Intel TDX Module before slicing, but times out on them after slicing. In total the slicing of HARNESSFORGE increases the sum of correct verifier results from 431 to 519 across all verifiers and tasks.

Threats to Validity. We are confident in the internal validity of our results. The created verification tasks were peer-reviewed as part of SV-COMP 2026 and we used best practices [10] for reliable and reproducible measurements. Regarding external validity, we are confident that our selection of verifiers represents a wide sample of the state of the art in software verification, but results may differ for other software projects.

4 Conclusion and Future Work

We described HARNESSFORGE, a tool that addresses the practical challenge of assembling standard verification tasks from industry-scale, multi-file C projects. The tool has been successfully applied to three practically relevant projects, creating 949 verification tasks. We envision that HARNESSFORGE will help practitioners to preprocess their code and apply software verifiers with ease.

We continue to develop HARNESSFORGE with our industry partners in three directions: First, we aim to reduce complexity in the verification task by slicing preconditions that only initialize or constrain data structures not used in the method under verification. Second, automatic generation of boilerplate code is planned to simplify the creation of verification harnesses. Third, we intend to assist engineers in writing correct and meaningful harnesses by providing a data-flow-based linter that targets common mistakes.

Data-Availability Statement. Both HARNESSEFORGE ([gitlab.com/sosy-lab/software/harnessforge](https://github.com/sosy-lab/software/harnessforge)) and the generated verification tasks for AWS C Common, GNU Coreutils project, and Intel TDX are available open source and archived at Zenodo [7].

Funding Statement. This project was funded by the Deutsche Forschungsgemeinschaft (DFG) under grants 378803395 (ConVeY) and 536040111 (Bridge), and a research gift from Intel.

References

- [1] Amazon.com. 2026. One Line Scan GitHub Repository. <https://github.com/aws-labs/one-line-scan>. Accessed: 2026-01-22.
- [2] P. Ayaziová, M. Jonáš, V. Mihalkovič, and J. Sedláček. 2026. Symbiotic: Submission to SV-COMP 2026. Zenodo. doi:10.5281/zenodo.17698724
- [3] D. Baier, D. Beyer, P.-C. Chien, M.-C. Jakobs, M. Jankola, M. Kettl, N.-Z. Lee, T. Lemberger, M. Lingsch-Rosenfeld, H. Wachowitz, and P. Wendler. 2024. Software Verification with CPAchecker 3.0: Tutorial and User Guide. In *Proc. FM (LNCS 14934)*. Springer, 543–570. doi:10.1007/978-3-031-71177-0_30
- [4] D. Baier, D. Beyer, P.-C. Chien, M. Jankola, M. Kettl, N.-Z. Lee, T. Lemberger, M. Lingsch-Rosenfeld, M. Spiessl, H. Wachowitz, and P. Wendler. 2024. CPAchecker 2.3 with Strategy Selection (Competition Contribution). In *Proc. TACAS (3) (LNCS 14572)*. Springer, 359–364. doi:10.1007/978-3-031-57256-2_21
- [5] D. Beyer. 2026. Evaluating Tools for Automatic Software Testing (Report on Test-Comp 2026). In *Proc. FASE (LNCS 16504)*. Springer, 449–468. doi:10.1007/978-3-032-22774-4_23
- [6] D. Beyer, P.-C. Chien, B.-Y. Huang, N.-Z. Lee, and T. Lemberger. 2026. A Case Study in Firmware Verification: Applying Formal Methods to Intel® TDX Module. In *Proc. TACAS (LNCS)*. Springer. doi:10.1007/978-3-032-22749-2_3
- [7] D. Beyer, P.-C. Chien, B.-Y. Huang, N.-Z. Lee, and T. Lemberger. 2026. Reproduction Package for FSE 2026 Submission ‘HarnessForge: Automated Extraction of Verification Tasks from Industry-Scale Software Projects’. Zenodo. doi:10.5281/zenodo.18348148
- [8] D. Beyer, P.-C. Chien, and M. Jankola. 2024. BENCHCLOUD: A Platform for Scalable Performance Benchmarking. In *Proc. ASE. ACM*, 2386–2389. doi:10.1145/3691620.3695358
- [9] D. Beyer, M. Dangl, and P. Wendler. 2018. A Unifying View on SMT-Based Software Verification. *J. Autom. Reasoning* 60, 3 (2018), 299–335. doi:10.1007/s10817-017-9432-6
- [10] D. Beyer, S. Löwe, and P. Wendler. 2019. Reliable Benchmarking: Requirements and Solutions. *Int. J. Softw. Tools Technol. Transfer* 21, 1 (2019), 1–29. doi:10.1007/s10009-017-0469-y
- [11] D. Beyer and J. Strejček. 2026. Evaluating Software Verifiers for C, Java, and SV-LIB (Report on SV-COMP 2026). In *Proc. TACAS (2) (LNCS 16506)*. Springer, 461–501. doi:10.1007/978-3-032-22749-2_23
- [12] D. Beyer and P. Wendler. 2026. CPAchecker 4.2.2. Zenodo. doi:10.5281/zenodo.1777566
- [13] R. Brunner, R. Dyer, M. Paquin, and E. Sherman. 2020. PAClab: A program analysis laboratory. In *Proc. ESEC/FSE. ACM*, 1616–1620. doi:10.1145/3368089.3417936
- [14] F. Busse, P. M. Gharat, C. Cadar, and A. F. Donaldson. 2022. Combining static analysis error traces with dynamic symbolic execution (experience paper). In *Proc. ISSA. ACM*, 568–579. doi:10.1145/3533767.3534384
- [15] C. Cadar, D. Dunbar, and D. R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. OSDI. USENIX Association*, 209–224. <https://dl.acm.org/doi/10.5555/1855741.1855756>
- [16] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2020. Code-Level Model Checking in the Software Development Workflow. In *Proc. ICSE (ICSE-SEIP '20)*. ACM, 11–20. ISBN: 9781450371230 doi:10.1145/3377813.3381347
- [17] E. M. Clarke, D. Kröning, and F. Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. TACAS (LNCS 2988)*. Springer, 168–176. doi:10.1007/978-3-540-24730-2_15
- [18] SV-COMP Community. 2026. AWS C Common in the SV-Benchmarks of SV-COMP 2026. <https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp26/c/aws-c-common>. Accessed: 2026-03-24.
- [19] SV-COMP Community. 2026. Coreutils v9.5 in the SV-Benchmarks of SV-COMP 2026. <https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp26/c/coreutils-v9.5-units>. Accessed: 2026-03-24.
- [20] SV-COMP Community. 2026. Intel TDX Module in the SV-Benchmarks of SV-COMP 2026. <https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp26/c/intel-tdx-module>. Accessed: 2026-03-24.
- [21] Intel Corporation. 2026. Intel Trust Domain Extensions. <https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html>. Accessed: 2026-04-01.
- [22] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. 2012. FRAMA-C. In *Proc. SEFM. Springer*, 233–247. doi:10.1007/978-3-642-33826-7_16
- [23] D. Dietsch, M. Bentele, M. Ebbinghaus, M. Heizmann, D. Klumpp, A. Podelski, and F. Schüssele. 2026. Ultimate Automizer SV-COMP 2026. Zenodo. doi:10.5281/zenodo.17735224
- [24] M. R. Gadelha, H. I. Ismail, and L. C. Cordeiro. 2017. Handling Loops in Bounded Model Checking of C Programs via *k*-induction. *Int. J. Softw. Tools Technol. Transf.* 19, 1 (February 2017), 97–114. doi:10.1007/s10009-015-0407-9
- [25] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. 2018. ESBMC 5.0: An Industrial-Strength C Model Checker. In *Proc. ASE. ACM*, 888–891. doi:10.1145/3238147.3240481
- [26] M. Heizmann, M. Bentele, D. Dietsch, X. Jiang, D. Klumpp, F. Schüssele, and A. Podelski. 2024. ULTIMATE AUTOMIZER and the Abstraction of Bitwise Operations (Competition Contribution). In *Proc. TACAS (3) (LNCS 14572)*. Springer, 418–423. doi:10.1007/978-3-031-57256-2_31
- [27] M. Heizmann, J. Hoenicke, and A. Podelski. 2013. Software Model Checking for People Who Love Automata. In *Proc. CAV (LNCS 8044)*. Springer, 36–52. doi:10.1007/978-3-642-39799-8_2
- [28] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. 2004. Abstractions from proofs. In *Proc. POPL. ACM*, 232–244. doi:10.1145/964001.964021
- [29] M. Jonáš, K. Kumor, J. Novák, J. Sedláček, M. Trtík, L. Zaoral, P. Ayaziová, and J. Strejček. 2024. SYMBIOTIC 10: Lazy Memory Initialization and Compact Symbolic Execution (Competition Contribution). In *Proc. TACAS (3) (LNCS 14572)*. Springer, 406–411. doi:10.1007/978-3-031-57256-2_29
- [30] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. 2019. Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer. In *Proc. VSTTE (LNCS 12031)*. Springer, 1–18. doi:10.1007/978-3-030-41600-3_1
- [31] D. Kroening and E. Clarke. 2025. CBMC 6.6.0. <https://github.com/diffblue/cbmc/releases/tag/cbmc-6.6.0>. Accessed: 2026-01-22.
- [32] X. Li. 2026. ESBMC submission for the pre-run of SV-COMP'26. Zenodo. doi:10.5281/zenodo.17741226
- [33] M. Milanese, R. Monat, A. Ouadjaout, and A. Miné. 2026. Mopsa at SV-COMP 2026. Zenodo. doi:10.5281/zenodo.17696794
- [34] C. Moloney, R. Dyer, and E. Sherman. 2026. Demonstrating ARG-V's Generation of Realistic Java Benchmarks for SV-COMP. In *Proc. TACAS (2) (LNCS 16506)*. Springer. doi:10.1007/978-3-032-22749-2_14
- [35] R. Monat, A. Ouadjaout, and A. Miné. 2025. MOPSA-C with Trace Partitioning and Autosuggestions (Competition Contribution). In *Proc. TACAS (3) (LNCS 15698)*. Springer, 229–235. doi:10.1007/978-3-031-90660-2_17
- [36] L. Nagy. 2026. Bear GitHub Repository. <https://github.com/rizotto/Bear>. Accessed: 2026-01-22.
- [37] J. Slabý, J. Strejček, and M. Trtík. 2012. Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution. In *Proc. FMICS (LNCS 7437)*. Springer, 207–221. doi:10.1007/978-3-642-32469-7_14
- [38] T. Wu, X. Li, E. Manino, R. Menezes, M. Gadelha, S. Xiong, N. Tihanyi, P. Petoumenos, and L. Cordeiro. 2025. ESBMC v7.7: Efficient Concurrent Software Verification with Scheduling, Incremental SMT and Partial Order Reduction (Competition Contribution). In *Proc. TACAS (3) (LNCS 15698)*. Springer, 223–228. doi:10.1007/978-3-031-90660-2_16