2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

A Data Set of Program Invariants and Error Paths

Dirk Beyer
LMU Munich, Germany

***Abstract*—The analysis of correctness proofs and counterexamples of program source code is an important way to gain insights into methods that could make it easier in the future to find invariants to prove a program correct or to find bugs. The availability of high-quality data is often a limiting factor for researchers who want to study real program invariants and real bugs. The described data set provides a large collection of concrete verification results, which can be used in research projects as data source or for evaluation purposes. Each result is made available as verification witness, which represents either program invariants that were used to prove the program correct (correctness witness) or an error path to replay the actual bug (violation witness). The verification results are taken from actual verification runs on 10 522 verification problems, using the 31 verification tools that participated in the 8th edition of the International Competition on Software Verification (SV-COMP). The collection contains a total of 125 720 verification witnesses together with various meta data and a map to relate a witness to the C program that it originates from. Data set is available at: https://doi.org/10.5281/zenodo.2559175**

***Index Terms*—Invariant Mining, Program Comprehension, Formal Verification, Model Checking, Program Analysis, Verification Witnesses, Program Invariants, Error Paths, Bugs**

I. Introduction

Automatic software verification is the problem of answering, for a given program and a given specification, the question: “Does the program fulfill the specification.” The answer is either TRUE or FALSE, or —because the problem is undecidable [43]— the verification tool runs out of resources (reports UNKNOWN). The past two decades were filled with breakthroughs in software verification, and model-checking technology is already used in practice [2], [29]. The maturity of the research area is showcased by the annual International Competition on Software Verification (SV-COMP)[1], which is a large-scale comparative evaluation where the world’s best automatic verifiers compete in solving software-verification problems without user interaction.

One of the most important problems in software verification is to find program invariants that help proving the program correct, and in software testing, to find a path through the program that witnesses a violation of the specification. There are many open questions in the field that could be answered using data-mining techniques. Invariant synthesis is a large research area and there are many successful techniques. However, the recent progress in data science opens new possibilities that might contribute to software verification and to better understanding of program invariants and error paths.

Since a few years, it is an established standard for software-verification tools to store the verification results, i.e., the program invariants and error paths, in an exchangeable format, called verification witness. The feature of storing verification witnesses was introduced in order to be able to validate verification results, to avoid spending time on investigating erroneous proofs and false alarms, and to increase the trustworthiness of verification technology. One of the standard solutions is to view software verification as a certifying algorithm [34], that is, to validate concrete answers of the algorithm using a witness. Applying this idea to software verification means that a verification tool does not only report a result from {TRUE, FALSE, UNKNOWN}, but in addition a witness that contains information to validate (hopefully with less resources) that the result of the verification process is correct. This possibility was investigated for bugs (“violation witnesses”, [6], [13], [17]) and for correctness proofs (“correctness witnesses”, [7], [12]).

This data set [10] provides a large collection of verification witnesses together with meta information and a mapping from the programs [9] for which the witnesses were produced.

II. Data Source, Methodology to Gather the Data

The data source from which the verification witnesses were produced is the largest and most diverse publicly known repository of verification tasks[2], which is used by the software-verification community to benchmark their tools. The benchmark repository contains thousands of programs derived from the Linux kernel, from BusyBox, models of the Secure Shell implementation, and also artificially created programs to explore features of verification tools. There are verification tasks of various sizes, from less than 10 lines up to hundreds of thousands of lines of code. The ‘difficulty’ of a verification problem does not depend on the length of the program; it is rather important for a benchmark suite to cover code that is similar to the code written in industrial practice. The competition on software verification [8] uses the verification problems from this repository. According to the rules of the competition, all verifiers need to write the verification results as exchangeable witnesses (witness automata in XML format).

The verification results (i.e., the witnesses) were produced using the verifiers that participated in the competition on the verification tasks that were used in the competition. The benchmarking (controlled execution) was done with BenchExec[3], a lightweight tool for container-based execution of tools and for results collection. All verifiers are available in an open repository[4]. The setup and technical configuration for these

[1]https://sv-comp.sosy-lab.org, [5]

[2]https://github.com/sosy-lab/sv-benchmarks, [9]

[3]https://github.com/sosy-lab/benchexec, [16]

[4]https://gitlab.com/sosy-lab/sv-comp/archives-2019/tree/svcomp19/2019

```
{
  "architecture": "32bit",
  "creationtime": "2018-12-05T12:59:54+01:00",
  "producer": "CPAchecker␣1.7-svn␣29852",
  "program-sha256": "d12be2991167702d1ce44aa0d0c4a2533945107abce0ec58bc87a3ffa57c4388",
  "programfile": "../../sv-benchmarks/c/loop-invgen/apache-get-tag_true-unreach-call_true-termination.i",
  "programhash": "d12be2991167702d1ce44aa0d0c4a2533945107abce0ec58bc87a3ffa57c4388",
  "sourcecodelang": "C",
  "specification": "CHECK(␣init(main()),␣LTL(G␣!␣call(__VERIFIER_error()))␣)",
  "witness-file": "witnessFileByHash/8481a3c9e45d6ccad5c5f8f4e5b9a00f218e30[...].graphml",
  "witness-sha256": "8481a3c9e45d6ccad5c5f8f4e5b9a00f218e30bbc4edd95f55fbfb695f5665f7",
  "witness-size": 17133,
  "witness-type": "correctness_witness"
}
```
Fig. 1: Example JSON record for a verification witness, `[...]` is used to abbreviate a long hash

verifiers, that is, all settings required to reproduce the data set, are documented in the benchmarks definitions.[5]

While the data set can be reproduced, it requires a large amount of computing resources: reproducing the data set would currently require a total of over 450 days of CPU time. The data set was computed on a compute cluster with 168 machines, each having one Intel Xeon E3-1230 v5 CPU, with 8 processing units each, a frequency of 3.4 GHz, 33 GB of RAM, and a GNU/Linux operating system (x86_64-linux, Ubuntu 18.04 with Linux kernel 4.15).

III. Structure of the Data Set

The data set is freely available at Zenodo[6] [10] as a ZIP archive that contains three directories:

`witnessFileByHash`. This directory contains 125 720 verification witnesses (of type correctness witness or violation witness). Each witness in this directory is stored in a file whose name is the SHA2 256-bit hash of its contents followed by the filename extension `.graphml`. The format of each witness is a standard XML-based exchange format[7], more precisely, witness automata in GraphML [18]. The witness contains also metadata in order to relate it to the verification problem for which it was produced. As example, let us consider the following witness file: **`8481a3c9e45d6ccad5c5f8f4e5b9a00f218e30[...].graphml`** (hyperlinks are typeset in blue color, `[...]` abbreviates a long strings). The file contains a correctness witness that was produced by the verifier CPAchecker, version 1.7-svn 29852, for the competition (sub-) category *ReachSafety-Loops*. More results for this category and by this verifier are available on the competition web site.[8] The witness was produced for the program **`loop-invgen/apache-get-tag_-true-unreach-call_true-termination.i`** from the open-source `sv-benchmarks` repository.[9] The invariants are contained in XML tags `<data key="invariant">`.

`witnessInfoByHash`. This directory contains for each witness in directory `witnessFileByHash` a record in JSON format (also using the SHA2 256-bit hash of the witness as filename, with `.json` as filename extension). In order to make the access to the metadata more convenient (avoid parsing all the witness files), the following metadata are provided in the JSON records: the bit architecture that the program was written for, the creation time, the producing verification tool, the program file path in order to find the program in the repository, the program hash to identify the exact version of the program, the programming language in which the program was written, the specification that was given as input for the verification when the witness was produced (using the specification language of the competition[10]), and some characteristics of the witness file itself (filename, SHA2 256-bit hash, size in bytes, and type of witness). As example, Fig. 1 shows the complete JSON record with metadata for the above-mentioned correctness witness: **`8481a3c9e45d6ccad5c5f8f4e5b9a00f218e30[...].json`**.

`witnessListByProgramHashJSON`. For convenient access to all witnesses for a certain program, this directory represents a function that maps each program (via its SHA2 256-bit hash) to a set of verification results (JSON records for witnesses as described above) that the verifiers have produced for that program. For each program for which witnesses exist, the directory contains a JSON file (using the SHA2 256-bit hash of the program as filename, with `.json` as filename extension) that contains all JSON records for witnesses for that program. As example, let us consider the program for which the above-mentioned witness was produced. The file **`d12be2991167702d1ce44aa0d0c4a253394510[...].json`** contains the JSON records for all witnesses produced for program **`loop-invgen/apache-get-tag[...]`**. This mapping and the metadata records are also used to generate information pages about witnesses that are linked from the competition web site.[11]

IV. Statistical Description of the Data Set

Table I reports aggregated numbers for some measures that characterize the witnesses in the data set. The table considers all witnesses that were produced by verifiers and could be parsed (witnesses produced in validation mode and syntactically invalid witnesses were excluded; we were not

[5]https://github.com/sosy-lab/sv-comp/tree/svcomp19/benchmark-defs

[6]https://doi.org/10.5281/zenodo.2559175

[7]https://github.com/sosy-lab/sv-witnesses

[8]https://sv-comp.sosy-lab.org/2019/results/results-verified/cpa-seq.2018-12-05_0546.results.sv-comp19_prop-reachsafety.ReachSafety-Loops.xml.bz2.merged.xml.bz2.table.html

[9]https://github.com/sosy-lab/sv-benchmarks/blob/svcomp19/c/loop-invgen/apache-get-tag_true-unreach-call_true-termination.i

[10]https://sv-comp.sosy-lab.org/2019/rules.php, [8]

[11]https://sv-comp.sosy-lab.org/inspect.php?programSHA256=d12be2991[...]

TABLE I: Statistical description of the data set of verification witnesses, numbers are rounded to 2 significant digits

Witness Measure	All Witnesses				Correctness Witnesses				Violation Witnesses			
	Median	Mean	Max	Sum	Median	Mean	Max	Sum	Median	Mean	Max	Sum
Number of States	27	950	$1.5 \cdot 10^6$	$58 \cdot 10^6$	23	1 100	$1.0 \cdot 10^6$	$39 \cdot 10^6$	31	750	$1.5 \cdot 10^6$	$19 \cdot 10^6$
Number of Transitions	27	1 200	$1.5 \cdot 10^6$	$74 \cdot 10^6$	24	1 400	$0.90 \cdot 10^6$	$52 \cdot 10^6$	31	860	$1.5 \cdot 10^6$	$22 \cdot 10^6$
Number of Invariants					3.0	380	$0.70 \cdot 10^6$	$3.1 \cdot 10^6$				
Length of All Invariants					270	35 000	$9.6 \cdot 10^6$	$290 \cdot 10^6$				

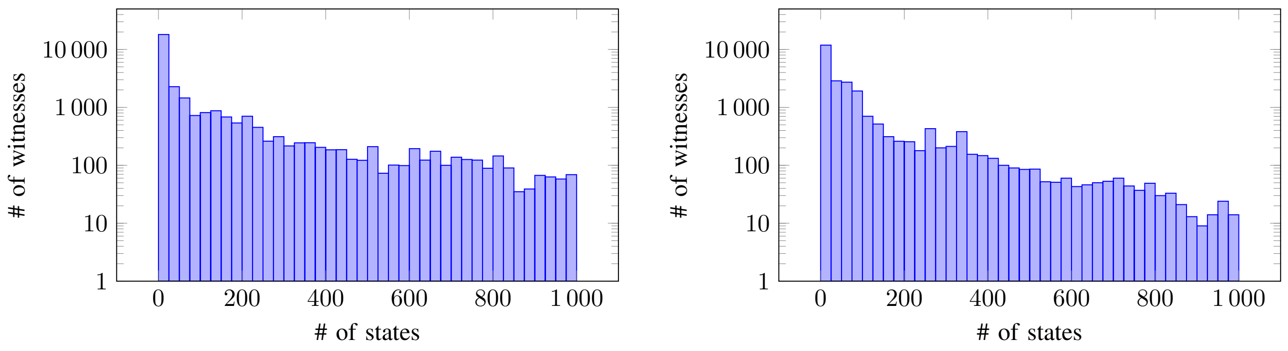

Fig. 2: Distribution of the number of states for correctness witnesses (left) and violation witnesses (right); histogram with 40 bins for the value range [0, 1 000]

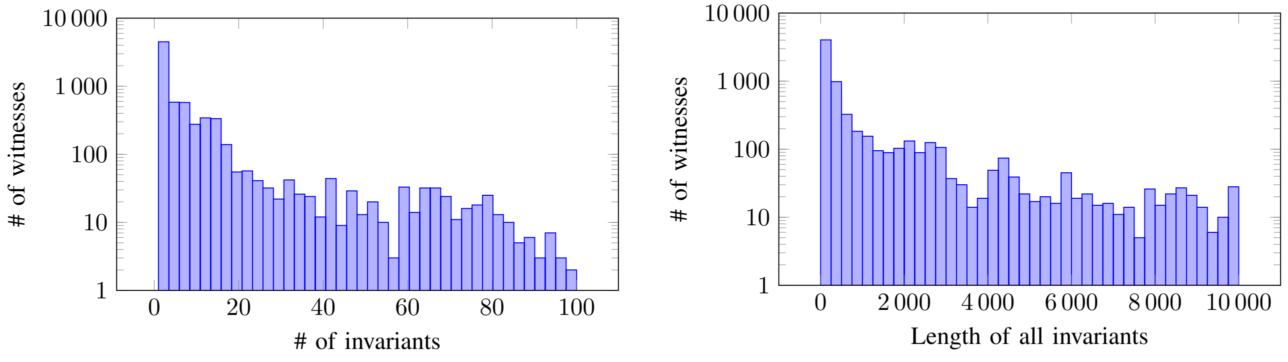

Fig. 3: Distribution of the number of invariants (left, value range [1, 100]) and the length of all invariants (right, value range [1, 10 000]) for correctness witnesses; histogram with 40 bins

able to parse two of the correctness witnesses due to their size and excluded them from all statistics: `2a6fe50[...].graphml` and `d4af9f2[...].graphml`, both produced by CBMC, both 954 MB large). The columns are grouped into values for (1) all witnesses, for (2) correctness witnesses, and for (3) violation witnesses. In each group, there are columns for the statistical properties median, arithmetic mean, maximum, and total sum. For each witness, we measure the number of states and the number of transitions of the witness automaton. For each correctness witness, we measure in addition the number of invariants and the length of all invariants of the witness. Those witnesses that have value zero for the last two measures are excluded from the statistics in the third and forth row.

In order to provide a rough picture of the size (in terms of states of the witness automata), we show the distribution of the number of states as histogram in Fig. 2. Verification witnesses can be very large; the following correctness witness is 295 MB large and has small invariants: `0e2805f6717240f656788d0d12ebeea0bbf473[...].graphml`. This correctness witness was constructed by the verifier CBMC (bounded model checker) for the following program: `zonotope_2_true-unreach-call_true-termination.c`.

In contrast to the above program, there are also programs for which the invariants can become pretty large, depending on how the verifier constructs the correctness proof. For example, consider the invariants in the following correctness witness: `7e7f927bbcb7d659c63aa4281a6201569a4b37[...].graphml`. The witness was constructed by CPACHECKER for the program: `gauss_sum_true-unreach-call_true-termination.i`. The distributions of the values for the measures number of invariants and length of all invariants of a witness are shown in Fig. 3. Although the witnesses do not need to contain full proofs (compare to proof-carrying code [35]), the individual invariants can still be quite large.

TABLE II: Statistical description of correctness witnesses, per verifier, numbers are rounded to 2 significant digits

Verifier	Number of States				Number of Invariants				Length All Invariants				Total
	Median	Mean	Max	Sum	Median	Mean	Max	Sum	Median	Mean	Max	Sum	Witnesses
2LS [41]	310	4 900	360 000	10 000 000	1.0	1.2	8	610	1 300	1 700	4 600	830 000	2 100
CBMC [30]	45	3 800	1 000 000	8 700 000	24	1 300	700 000	3 100 000	640	21 000	9 000 000	48 000 000	2 300
CPA-BAM-BnB [1]	440	2 000	51 000	4 400 000	3.0	5.6	130	4 000	130	220	12 000	160 000	2 200
CPA-Seq [23]	140	760	21 000	3 500 000	2.0	4.8	340	8 100	260	84 000	9 400 000	140 000 000	4 600
PeSCo [38]	130	580	14 000	1 700 000	3.0	8.5	420	6 500	4 000	130 000	9 600 000	97 000 000	2 900
Skink [19]	2.0	2.0	2	690	1.0	1.0	1	350	1.0	1.0	1	350	350
UAutomizer [27]	310	1 900	61 000	6 700 000	2.0	6.0	79	3 500	180	1 300	160 000	750 000	3 600
UKojak [36]	310	760	17 000	1 100 000	1.0	1.9	32	1 400	59	660	120 000	470 000	1 400
UTaipan [26]	230	1 400	61 000	2 900 000	2.0	4.4	68	1 300	120	780	22 000	220 000	2 100
VeriAbs [22]	1.0	30	790	32 000	4.0	5.2	18	560	470	2 400	65 000	260 000	1 100
VIAP [37]	2.0	52	2 300	11 000	1.0	1.0	1	170	1.0	1.0	1	170	210

Both above-mentioned programs are small (both below 30 lines of code), thus, the number and size of the invariants in the correctness witnesses suggest that there is low correlation between program size and verification effort (at least for the verifiers that constructed the witnessed correctness proofs).

Table II reports more details on correctness witnesses. The table contains rows only for those verifiers that produced at least one correctness witness with an invariant. The statistical properties are the same as before and exclude zero-values again; the last column shows the total number of produced correctness witnesses for each of those verification tools.

V. Future Research Questions

In the following we outline a few example applications of program invariants and error paths. The data set could be used as data source or for evaluation.

Visualization of error paths. Violation witnesses can help to understand a bug by replying the error path and inspect the error path with visualization tools [11]. Also, violation witnesses can be used to generate test cases that execute the error path described by the witness [14]. Future research projects might develop an infrastructure that provides engineers with a convenient way to integrate witness-based results validation into their daily workflow.

Annotation of Programs with Invariants. Understanding of a program could be improved by source-code editors that show loop invariants for the loop heads (e.g., via tool tip when the developer hovers the mouse pointer over the code). Also, programs could be transformed in order to annotate invariants as comments in ACSL format [4].

Classification of Bugs. It can be suspected that there exist classes of bugs that share similarities in their characteristics. Using machine learning and clustering technology, one can automatically and semi-automatically find classes of similar witnesses and try to understand the characteristics of the bugs.

Classification of Program Invariants. A correctness witness can contain the (loop) invariants that are a necessary to prove the correctness of the program. It is not yet studied what kind of theories are used in those invariants and how expressive the invariants are. For example, are the invariants mostly of the form $x = y$, how much linear arithmetics is involved, how many variables per invariant? Are the invariants inequalities or equalities, what is their structure?

For all these ideas, a large number of witnesses is necessary to evaluate ideas and research approaches.

VI. Potential Improvements, Limitations

The research area of software verification is rapidly growing, and the repository of verification tasks is expected to grow as well, and therefore, further invariants and error paths can be produced in the future. As new tasks will be more difficult to prove, future verification runs might produce more complex and more interesting invariants.

The data set is limited to the C programming language. All conclusions drawn from this collection of witnesses might be limited to C programs. However, there is no principle limitation: witnesses and results validation are not limited to a certain programming language, and the insights from studying invariants and bugs can be of general interest.

The data set is also limited by the verifiers. The best publicly known verifiers participated in the competition, but in principle there might be other verification tools that can generate more interesting invariants.

VII. Related Work

This work builds on the literature on witness-based results validation [6], [7], [12], [13], [17] and on the verification tools that participated in the Competition on Software Verification (SV-COMP) 2019 [1], [3], [15], [19]–[28], [30]–[33], [36]–[42]. See also the SV-COMP competition report [8].

VIII. Conclusion

The presented data set provides a large collection of verification witnesses, which are containers for program invariants and error paths in an exchangeable format. This collection of witnesses for many different kinds of programs can serve as data source to apply clustering, machine learning, matching, and other classification techniques in order to gain new insights about the program invariants and bugs on the semantic level.

We hope that researchers might find it interesting to work with verification witnesses towards deeper semantical understanding of the nature of correctness proofs and software bugs. Producing the witnesses in the data set required 4 days of CPU time on a compute cluster with 168 CPUs.

References

[1] 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. pp. 355–359. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_22

[2] Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Proc. POPL. pp. 1–3. ACM (2002). https://doi.org/10.1145/503272.503274

[3] 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)

[4] Baudin, P., Cuoq, P., Fillâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language Version 1.10. http://frama-c.com/download/acsl.pdf

[5] 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

[6] 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

[7] Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS. pp. 331–349. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_20

[8] Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3). LNCS 11429, Springer (2019)

[9] Beyer, D.: SV-Benchmarks: Benchmark set of 8th Intl. Competition on Software Verification (SV-COMP 2019). Zenodo (2019). https://doi.org/10.5281/zenodo.2598729

[10] Beyer, D.: Verification witnesses from SV-COMP 2019 verification tools. Zenodo (2019). https://doi.org/10.5281/zenodo.2559175

[11] Beyer, D., Dangl, M.: Verification-aided debugging: An interactive web-service for exploring error witnesses. In: Proc. CAV (2). pp. 502–509. LNCS 9780, Springer (2016). https://doi.org/10.1007/978-3-319-41540-6_28

[12] 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

[13] 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

[14] 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

[15] Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16

[16] Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Proc. SPIN. pp. 160–178. LNCS 9232, Springer (2015). https://doi.org/10.1007/978-3-319-23404-5_12

[17] Beyer, D., Wendler, P.: Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In: Proc. SPIN. pp. 1–17. LNCS 7976, Springer (2013). https://doi.org/10.1007/978-3-642-39176-7_1

[18] Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report. In: Graph Drawing. pp. 501–512. LNCS 2265, Springer (2001). https://doi.org/10.1007/3-540-45848-4_59

[19] Cassez, F., Sloane, A.M., Roberts, M., Pigram, M., Suvanpong, P., de Aledo Marugán, P.G.: Skink: Static analysis of programs in LLVM intermediate representation (competition contribution). In: Proc. TACAS. pp. 380–384. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_27

[20] Chalupa, M., Strejcek, 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

[21] Chalupa, M., Vitovská, M., Strejcek, J.: Symbiotic 5: Boosted instrumentation (competition contribution). In: Proc. TACAS. pp. 442–446. LNCS 10806, Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_29

[22] Chimdyalwar, B., Darke, P., Chauhan, A., Shah, P., Kumar, S., Venkatesh, R.: VeriAbs: Verification by abstraction (competition contribution). In: Proc. TACAS. pp. 404–408. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_32

[23] Dangl, M., Löwe, S., Wendler, P.: CPACHECKER with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34

[24] Gadelha, M.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC v6.0: Verifying C programs using k-induction and invariant inference (competition contribution). In: Proc. TACAS. LNCS 11429, Springer (2019)

[25] 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 (Feb 2017). https://doi.org/10.1007/s10009-015-0407-9

[26] Greitschus, M., Dietsch, D., Heizmann, M., Nutz, A., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Taipan: Trace abstraction and abstract interpretation (competition contribution). In: Proc. TACAS. pp. 399–403. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_31

[27] Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Nutz, A., Musa, B., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Automizer with an on-demand construction of Floyd-Hoare automata (competition contribution). In: Proc. TACAS. pp. 394–398. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_30

[28] Khazem, K., Tautschnig, M.: CBMC Path: A symbolic execution retrofit of the C bounded model checker (competition contribution). In: Proc. TACAS. LNCS 11429, Springer (2019)

[29] Khoroshilov, A.V., Mutilin, V.S., Petrenko, A.K., Zakharov, V.: Establishing Linux driver verification process. In: Proc. Ershov Memorial Conference. pp. 165–176. LNCS 5947, Springer (2009). https://doi.org/10.1007/978-3-642-11486-1_14

[30] Kröning, D., Tautschnig, M.: CBMC: C bounded model checker (competition contribution). In: Proc. TACAS. pp. 389–391. LNCS 8413, Springer (2014)

[31] Lauko, H., Ročkai, P., Barnat, J.: Symbolic computation via program transformation. In: Proc. ICTAC. pp. 313–332. Springer (2018)

[32] Lauko, H., Štill, V., Ročkai, P., Barnat, J.: Extending DIVINE with symbolic verification using SMT (competition contribution). In: Proc. TACAS. LNCS 11429, Springer (2019)

[33] Malik, V., Hruska, M., Schrammel, P., Vojnar, T.: 2LS: Heap analysis and memory safety (competition contribution). Tech. Rep. abs/1903.00712, CoRR (2019)

[34] McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011). https://doi.org/10.1016/j.cosrev.2010.09.009

[35] Necula, G.C.: Proof-carrying code. In: Proc. POPL. pp. 106–119. ACM (1997). https://doi.org/10.1145/263699.263712

[36] 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

[37] Rajkhowa, P., Lin, F.: VIAP 1.1: Automated system for verifying integer assignment programs with loops (competition contribution). In: Proc. TACAS. LNCS 11429, Springer (2019)

[38] Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS. LNCS 11429, Springer (2019)

[39] Rocha, H., Ismail, H., Cordeiro, L.C., Barreto, R.S.: Model checking embedded C software using k-induction and invariants. In: Proc. SBESC. pp. 90–95. IEEE (2015). https://doi.org/10.1109/SBESC.2015.24

[40] Rocha, W., Rocha, H., Ismail, H., Cordeiro, L.C., Fischer, B.: DepthK: A k-induction verifier based on invariant inference for C programs (competition contribution). In: Proc. TACAS. pp. 360–364. LNCS 10206, Springer (2017). https://doi.org/10.1007/978-3-662-54580-5_23

[41] Schrammel, P., Kröning, D.: 2LS for program analysis (competition contribution). In: Proc. TACAS. pp. 905–907. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_56

[42] Still, V., Rockai, P., Barnat, J.: DIVINE: Explicit-state LTL model checker (competition contribution). In: Proc. TACAS. pp. 920–922. LNCS 9636, Springer (2016)

[43] Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. In: Proc. LMS. vol. s2-42, pp. 230–265. London Mathematical Society (1937). https://doi.org/10.1112/plms/s2-42.1.230