The Static Analyzer Infer in SV-COMP (Competition Contribution)

Matthias Kettl(✉) and Thomas Lemberger

LMU Munich, Germany

Abstract. We present INFER-SV, a wrapper that adapts INFER for SV-COMP. INFER is a static-analysis tool for C and other languages, developed by Facebook and used by multiple large companies. It is strongly aimed at industry and the internal use at Facebook. Despite its popularity, there are no reported numbers on its precision and efficiency. With INFER-SV, we take a first step towards an objective comparison of INFER with other SV-COMP participants from academia and industry.

1 Facebook Infer

INFER [6] is a compositional and incremental static-analysis tool developed at Facebook. INFER supports a wide array of analyses; this includes memory safety, buffer overruns, performance constraints and different reachability analyses for C, C++, Objective C, Java, C#, and .Net. For memory analysis, INFER uses bi-abduction [7] with separation logic [14]. INFER supports the integration of new abstract domains through the abstract-interpretation framework Infer:AI. INFER analyzes programs compositionally (building method summaries) and incrementally (only analyzing changed program parts). In contrast to most other tools that participate in SV-COMP, INFER is not an academic verifier. Instead, it is aimed at practical use during software development. This has direct implications on the development focus: When INFER is told to incrementally analyze software, it outputs only newly discovered bugs and does not re-report bugs found in previous analyses. This allows developers to ignore warnings not deemed relevant and reduces the cognitive burden on developers due to false alarms. Multiple large companies use INFER—among others: Amazon Web Services, Facebook, Microsoft, Mozilla, and Spotify. At the time of this writing, INFER has more than 12 000 stars on GitHub and was forked over 1 500 times. Despite its popularity, there are no reported numbers on INFER's precision and soundness. With the participation of INFER in the C language track of SV-COMP '22, we hope to take a first step towards an objective comparison of INFER with other verifiers.

The following other commercial verifiers participate in SV-COMP '22: 2LS [16], CBMC [10], CRUX [1], FRAMA-C [5], VERIABS [12], and VERIFUZZ [9].

[1] https://crux.galois.com/

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 451–456, 2022.
https://doi.org/10.1007/978-3-030-99527-0_30

2 Infer in SV-COMP

2.1 Infer-SV

Verification. We provide the wrapper Infer-sv to adapt Infer to the SV-COMP specification format for program properties. Infer-sv parses the property to analyze, adjusts the program under analysis for Infer, runs Infer with fitting analyses, and reports a verification verdict based on the feedback produced by Infer. Infer-sv supports the following SV-COMP program properties:

no-overflow. The aim is to check for arithmetic overflows on signed-integer types. Infer-sv runs Infer's buffer-overrun analysis [2] to detect these.

unreach-call. The aim is to check for reachable calls to function `reach_error`. Infer provides a function-call reachability analysis [3], but this analysis proved very imprecise. To mitigate this, Infer-sv performs a program transformation [4]: It replaces each call to function `reach_error` with an overflow-provoking statement `int __reach_error_x = 0x7fffffff + 1`. No task with property unreach-call contains a signed-integer overflow, so the original reachability property holds if and only if any of the introduced overflows is reachable. Infer-sv runs Infer's buffer-overrun analysis on the transformed program to check this.

valid-memsafety. The aim is to check for invalid pointer dereferences, invalid frees of memory, and memory leaks. To analyze memory safety, Infer-sv uses two analyses: bi-abduction [5] and Infer:Pulse [6]. SV-COMP requires verifiers to report the concrete type of violation detected: *valid-deref*, *valid-memtrack*, or *valid-free*. Infer-sv analyzes the error codes reported by Infer to determine the exact violation found. If Infer reports multiple fitting warnings, we take the first.

Witnesses. SV-COMP requires participants to report GraphML verification-result witnesses [3, 4] in tandem with each result, and these witnesses must be successfully validated by at least one participating witness validator. Natively, Infer does not support the generation of GraphML witnesses. To mitigate this, Infer-sv creates generic witnesses: When reporting a violation, it generates a violation witness [4] that represents all possible program paths. When reporting a program safe, it generates a correctness witness [3] that only contains the trivial invariant 'true'. These witnesses do not helpfully guide towards a violation or proof, but are valid according to the SV-COMP rules.

Participation. Infer-sv participates hors concours in the categories ReachSafety, ConcurrencySafety, NoOverflows, and SoftwareSystems. Because of missing support, we exclude Infer-sv from categories aimed at float handling, as well as category MemSafety-MemCleanup.

[2] https://fbinfer.com/docs/checker-bufferoverrun
[3] https://fbinfer.com/docs/checker-annotation-reachability
[4] https://github.com/facebook/infer/issues/763
[5] https://fbinfer.com/docs/checker-biabduction
[6] https://fbinfer.com/docs/checker-pulse

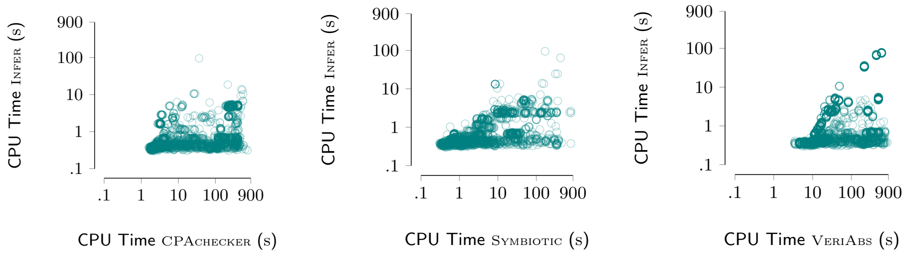

Fig. 1: Comparison of the run time (in CPU time seconds) of three SV-COMP '22 medalists and Infer, across all tasks correctly solved by the respective pair

```
int main() {
  if (0) {
    int x = 0x7fffffff + 1;
  }
}
```

(a) Infer correctly reports safety

```
void reach_error() {
  int x = 0x7fffffff + 1;
}
int main() {
  if (0) {
    reach_error();
  }
}
```

(b) Infer incorrectly reports an alarm

```
int main() {
  int x = 0x7fffffff;
  int y = -1;
  while (x > 0) {
    x = x - 2*y;
  }
}
```

(c) Infer correctly reports an alarm

```
int main() {
  int x = 0x7fffffff;
  int y = -1;
  while (x > 0) {
    x = x - 2*y;
    y = y + 2;
  }
}
```

(d) Infer incorrectly reports safety

Fig. 2: Examples of Infer's inconsistent results

2.2 Strengths of Infer

Infer scales well [6]. This shows in the SV-COMP results: For 6 000 out of 8 000 tasks with a verification verdict, Infer finishes the analysis in less than one second of CPU time. The remaining 2 000 tasks each take less than 100 s of CPU time. This means that Infer stays significantly below the time limit of 900 s per task. Figure 1 compares the run time of Infer (in CPU-time seconds) to the best SV-COMP '22 tools in the categories that Infer participated in: CPAchecker [11], Symbiotic [8], and VeriAbs [12]. Each plot shows the run time for all tasks that are correctly solved by both Infer and the respective other verifier (independent of result validation). It is visible that Infer (y-axis) is significantly faster than the other tools (x-axis) for almost all tasks. This speed makes Infer integrate well in continuous-integration development systems [13, 15].

2.3 Weaknesses of Infer

Infer demonstrates low analysis precision. Figures 2a and 2b illustrate a low precision across function calls (intraprocedural analysis): Both programs contain an unreachable, signed integer overflow. The only difference is the indirection in Fig. 2b due to the additional function call. Infer correctly reports Fig. 2a safe, but incorrectly reports an alarm for Fig. 2b. We assume that the intraprocedural analysis of Infer does not check whether `reach_error` is reachable from the program entry. Infer-sv mitigates this issue for property *unreach-call* through the mentioned program transformation, but this imprecision still leads Infer to report wrong alarms across all program properties.

Infer can also show imprecision within a single function. Consider Figs. 2c and 2d: The only change between Fig. 2c and Fig. 2d is the addition of a statement in line 6, `y = y + 2`. This has no influence on the integer overflow in line 5, so both programs contain an overflow. Infer correctly reports the overflow for Fig. 2c, but wrongly reports Fig. 2d safe.

These imprecisions strongly reflect in the SV-COMP results of Infer, leading to many incorrect proofs and alarms.

3 Usage

Infer-sv requires Python 3.6 or later. Script **setup.sh** downloads and extracts version 1.1.0 of Infer. From the tool's directory, Infer-sv can be run with the following command:

```
./infer-wrapper.py \
    --data-model {ILP32 or LP64} \
    --property path/to/property.prp \
    --program path/to/program.c \
```

Setting the data model is optional. Infer-sv will print the recognized property and the command line it uses to call Infer. Infer-sv prints the full output of Infer, including all warnings, and the final verification verdict on the last line. The verification verdict can be `true`, `false`, `unknown` or `error`.

4 Conclusion

The participation of Infer in SV-COMP allows an objective comparison with other verifiers for C. This shows that the selected analyses of Infer are very efficient, but suffer from strong imprecision on the considered benchmark tasks.

Contributors. Infer [7] is developed by Facebook and the open-source community under the MIT license, and Infer-sv [8] is developed under the Apache 2.0 license at the Software and Computational Systems Lab at LMU Munich, led by Dirk Beyer.

[7] https://github.com/facebook/infer
[8] https://gitlab.com/sosy-lab/software/infer-sv

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

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [1] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of our verifier as used in the competition is archived together with other participating tools [2].

References

1. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
2. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
3. 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
4. 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
5. Beyer, D., Spiessl, M.: The static analyzer FRAMA-C in SV-COMP (competition contribution). In: Proc. TACAS (2). Springer (2022)
6. Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Proc. NFM. pp. 3–11. LNCS 9058, Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_1
7. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. ACM **58**(6), 26:1–26:66 (2011). https://doi.org/10.1145/2049697.2049700
8. Chalupa, M., Řechtáčková, A., Mihalkovič, V., Zaoral, L., Strejček, J.: SYMBIOTIC 9: Parallelism and invariants (competition contribution). In: Proc. TACAS (2). Springer (2022)
9. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: Verifuzz: Program aware fuzzing - (competition contribution). In: Proc. TACAS, part 3. pp. 244–249. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22
10. 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
11. 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
12. 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
13. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019). https://doi.org/10.1145/3338112

14. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Proc. TACAS. LNCS, vol. 3920, pp. 287–302. Springer (2006). https://doi.org/10.1007/11691372_19
15. Harman, M., O'Hearn, P.W.: From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In: Proc. SCAM. pp. 1–23. IEEE (2018). https://doi.org/10.1109/SCAM.2018.00009
16. Malík, V., Schrammel, P., Vojnar, T.: 2LS: Heap analysis and memory safety (competition contribution). In: Proc. TACAS (2). pp. 368–372. LNCS 12079, Springer (2020). https://doi.org/10.1007/978-3-030-45237-7_22