GOBLINT: Abstract Interpretation for Memory Safety and Termination
(Competition Contribution)

Simmo Saan[1(✉)]*, Julian Erhard[2,3], Michael Schwarz[2], Stanimir Bozhilov[2], Karoliine Holter[1], Sarah Tilscher[2,3], Vesal Vojdani[1], and Helmut Seidl[2]

[1] University of Tartu, Tartu, Estonia
{simmo.saan,karoliine.holter,vesal.vojdani}@ut.ee
[2] Technische Universität München, Garching, Germany
{julian.erhard,m.schwarz,stanimir.bozhilov,
sarah.tilscher,helmut.seidl}@tum.de
[3] Ludwig-Maximilians-Universität München, Munich, Germany

Abstract. GOBLINT is an abstract interpreter of C programs, focusing on the analysis of multi-threaded code. It is equipped with a variety of abstract domains, as well as analyses which allow it to reason about an array of program properties in a highly configurable manner. GOBLINT has been extended with support for the detection of memory safety bugs and non-termination.

1 Verification Approach

GOBLINT is an abstract-interpretation–based static analyzer of C code, with an emphasis on the sound analysis of multi-threaded programs [14, 15]. It uses side-effecting constraint systems [2] to combine context-sensitive analysis of local states with flow-insensitive analysis of data possibly shared between threads. GOBLINT is equipped with a range of different analyses that, in turn, build on multiple abstract domains for expressing candidate program invariants.

1.1 Memory Safety

Techniques for detecting memory-related bugs have been extensively studied [6, 9, 10, 20]. While GOBLINT did not target such bugs in the past, new analyses for the *sound* analysis of memory safety have been added for SV-COMP 2024. The analyzer already tracks abstract address sets for pointer variables. A single abstract address consists of a variable and an abstract offset. The analyzer distinguishes between regular program variables and allocated memory blocks, which are identified by their respective allocation sites together with the allocating thread and possibly an allocation counter.

* Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 381–386, 2024.
https://doi.org/10.1007/978-3-031-57256-2_25

The new analyses are concerned with the detection of the following memory-safety bugs: *invalid memory deallocations*, *invalid pointer dereferences*, as well as *memory leaks*. Beyond *null-pointer dereferences*, two further kinds of invalid dereferences are now considered: *memory out-of-bounds* accesses and *use-after-free* (UAF) bugs. Memory out-of-bounds accesses can be uncovered by obtaining the size, as well as the offset from the base address of the memory being accessed. To determine whether an access via some offset may be out of bounds, the analysis relies on an expressive combination of integer domains including intervals.

Invalid dereferences due to use-after-frees can be detected in the single- and multi-threaded case. For the single-threaded setting, the analysis uses the allocation-site abstractions in order to keep track of potentially already deallocated memory, and warns on accesses to such memory. Regarding the multi-threaded case, it additionally leverages Goblint's side-effecting functionality by maintaining a global invariant that, for each piece of deallocated memory, collects the set of all threads that may free it. Goblint tracks abstract thread IDs which allow reasoning about which threads may run in parallel [17]. The may-happen-in-parallel (MHP) information from the abstract thread ID domain (and a dedicated analysis of thread joins) is used to infer whether an access to a piece of memory may happen in parallel with (or after) the deallocation of the same piece of memory by another thread. In addition, invalid frees due to possibly occurring double frees are flagged by this analysis as well.

Potential memory leaks can be detected thanks to a dedicated analysis. To this end, all allocated memory blocks are tracked path- and context-sensitively. Furthermore, the allocation counter is relied on to potentially exclude memory leaks for a particular allocation site. Calls to deallocating functions, such as `free`, have the effect of removing pieces of tracked (and now deallocated memory) from the state, whenever the analysis determines that the passed pointer *must* point to an abstract block of memory which describes a single concrete memory location. At all exit points of the program, it is then checked whether the set of possibly still allocated memory is empty. In case any such set is non-empty, a memory leak is reported. In the multi-threaded case, the analysis checks the following stronger property and warns whenever that property may be violated:

1. all threads have terminated at the end point of `main`, and
2. `exit` and similar functions, causing early termination, are not called, and
3. at its return, each thread has freed all the memory it allocated.

This property allows for a thread-modular analysis, where sets of allocated and freed memory are maintained in a flow- and context-sensitive manner.

We remark that the analysis for memory leaks tracks which heap-allocated memory *may not* be freed yet, while the analysis to detect UAF issues tracks which memory *may* potentially already be freed. One direction of improvement would be to consider tracking relational pointer information along the lines of Seidl et al. [18] and, additionally, consider relational information about the lengths of arrays and memory blocks. This may be useful in the case of variable length arrays and dynamically allocated memory for which the size is not statically known.

1.2 Termination

A termination analysis has been added, largely leveraging existing features of the framework. This highlights the versatility of the framework. To account for non-termination due to loops, a counter variable is inserted into each loop and incremented in every loop iteration. A relational polyhedra analysis based on APRON [8] is then used to determine whether the counter variable is bounded. To detect potential non-termination due to recursion, the notion of a call graph is enhanced by considering functions together with their respective abstract calling contexts and taking dynamic calls via pointers into account. This graph is *a posteriori* extracted out of the analysis result and then checked for cycles in a post-processing phase. In case no cycles (including self-loops) exist in the abstract call graph, there can be no cycles in the concrete call graph.

The currently implemented termination analysis is just a first step in the realization of related techniques. Future work may, e.g., be the tuning of the abstract contexts for this use-case, or the incorporation of more involved techniques for termination analysis by abstract interpretation [4, 5]. Extending the presented approaches to the non-termination of concurrent programs while remaining as thread-modular as possible seems particularly challenging.

2 Software Architecture

GOBLINT is implemented in ∼54,000 lines of OCAML and uses an updated fork of CIL [12] as its parser frontend for the C language. It depends on APRON [8] for relational analyses. No other major libraries or external tools are required.

The modular architecture of GOBLINT [1] allows a combination of analyses to be selected and automatically configured at runtime [15]. Analyses are defined through their abstract domains and transfer functions, which can communicate with other analyses using predefined queries and events. The combined analyses together with the control-flow graphs of the functions yield a side-effecting constraint system [2], which is solved using a local generic solver [19]. The solution is post-processed to determine the verdict and construct a witness.

3 Strengths and Weaknesses

GOBLINT once again demonstrated its soundness in this year's competition, i.e., it did not produce any false negatives. The only other tools that did not produce any false negatives are AISE [21] (competing only in *ReachSafety-Loops*), BRICK (competing in three sub-categories of *ReachSafety*), and MOPSA [11] (competing in all categories except *ConcurrencySafety* and *Termination*). GOBLINT is thus the only *sound* tool in SV-COMP 2024 to support *all* properties, and the only sound tool represented in the overall ranking. Among the tools participating in the overall ranking, GOBLINT, despite targeting only proofs – which are traditionally considered to be more time-consuming than finding counter-examples – leads the pack in terms of points achieved in ≤ 9 s. This is most pronounced when

considering runtimes ≤ 1 s. This highlights the efficiency of Goblint. Beyond these observations, we briefly discuss the newly added analyses here. Support for soundly detecting memory safety bugs greatly broadens the applicability of the analyzer, evidencing the flexibility of the underlying framework. Of particular note is the support for verifying the memory-safety of multi-threaded programs in a thread-modular way, yielding the second-best score in *ConcurrencySafety-MemSafety*, after Deagle [7]. Turning to termination analysis, the added analysis demonstrates that a considerable chunk of the SV-COMP benchmarks in this category can be handled by using our extended dynamic call graph to deal with recursion and ghost counters together with numerical relational domains to deal with loops. Finally, Goblint now comes with dedicated support for analyzing programs using `setjmp`/`longjmp` and flagging their misuse [16]. We have contributed programs using this language feature to the benchmark suite.

A general weakness of Goblint currently is that, while it supports expensive but expressive relational domains such as polyhedra, it lacks a heuristic when to activate them, and thus only uses them for termination analysis. Activating these domains based on some program properties, or attempting analysis with such expensive domains after an analysis without them was inconclusive, may help to improve the precision of the analyzer without compromising its efficiency.

4 Tool Setup and Configuration

Goblint version `svcomp24-0-gc2e9465a7` participated in SV-COMP 2024 [3, 13]. It is available in both binary (Ubuntu 22.04) and source code form at our GitHub repository.[4] Instructions for building from source can be found in the `README`. Both the tool-info module and the benchmark definition for SV-COMP are named `goblint`. They correspond to running the tool as follows:

```
./goblint --conf conf/svcomp24.json \
          --set ana.specification property.prp input.c
```

Goblint participated in *all* the categories, while opting-out from *FalsificationOverall*.

5 Software Project and Contributors

Goblint development takes place on GitHub, while related publications are listed on its website.[5] It is an MIT-licensed project initiated by Technische Universität München and the University of Tartu.

Acknowledgments. This work was supported by Deutsche Forschungsgemeinschaft (DFG) – 378803395/2428 ConVeY 2. We would like to thank everyone who has contributed to Goblint over the years, especially the students who contributed the termination analysis, namely: Thomas Lagemann, Johanna Franziska Schinabeck, Alexander Schlenga, and Isidor Zweckstetter.

[4] https://github.com/goblint/analyzer/releases/tag/svcomp24
[5] https://github.com/goblint/analyzer and https://goblint.in.tum.de

Data Availability Statement. All data of SV-COMP 2024 are archived as described in the competition report [3] and available on the competition website. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of GOBLINT as used in the competition is archived on Zenodo [13].

Bibliography

[1] Apinis, K.: Frameworks for analyzing multi-threaded C. Ph.D. thesis, Technische Universität München (2014)

[2] Apinis, K., Seidl, H., Vojdani, V.: Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In: APLAS '12, pp. 157–172, Springer (2012), DOI: 10.1007/978-3-642-35182-2_12

[3] Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: TACAS '24, Springer (2024)

[4] Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: POPL '12, pp. 245–258, ACM (2012), DOI: 10.1145/2103656.2103687

[5] Dimovski, A.S.: Lifted termination analysis by abstract interpretation and its applications. In: GPCE '21, pp. 96–109, ACM (2021), DOI: 10.1145/3486609.3487202

[6] Gui, B., Song, W., Xiong, H., Huang, J.: Automated use-after-free detection and exploit mitigation: How far have we gone? IEEE Trans. Software Eng. **48**(11), 4569–4589 (2022), DOI: 10.1109/TSE.2021.3121994

[7] He, F., Sun, Z., Fan, H.: DEAGLE: An SMT-based verifier for multi-threaded programs. In: TACAS '22, vol. 2, pp. 424–428, Springer (2022), DOI: 10.1007/978-3-030-99527-0_25

[8] Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: CAV '09, pp. 661–667, Springer (2009), DOI: 10.1007/978-3-642-02658-4_52

[9] Jones, J., Wasson, J., Brown, S., Poulsen, S., Aldous, P., Mercer, E.: Memory safety in C by abstract interpretation. SIGSOFT Softw. Eng. Notes **43**(4), 56 (2019), DOI: 10.1145/3282517.3282530

[10] Loginov, A., Yahav, E., Chandra, S., Fink, S., Rinetzky, N., Nanda, M.: Verifying dereference safety via expanding-scope analysis. In: ISSTA '08, pp. 213–224, ACM (2008), DOI: 10.1145/1390630.1390657

[11] Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: MOPSA-C: Improved verification for C programs, simple validation of correctness witnesses. In: TACAS '24, Springer (2024)

[12] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC '02, pp. 213–228, Springer (2002), DOI: 10.1007/3-540-45937-5_16

[13] Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint at SV-COMP 2024 (Nov 2023), DOI: 10.5281/zenodo.10202867, tool artifact

[14] Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: GOBLINT: Thread-modular abstract interpretation using side-effecting

constraints. In: TACAS '21, pp. 438–442, Springer (2021), DOI: 10.1007/978-3-030-72013-1_28
[15] Saan, S., Schwarz, M., Erhard, J., Pietsch, M., Seidl, H., Tilscher, S., Vojdani, V.: GOBLINT: Autotuning thread-modular abstract interpretation. In: TACAS '23, vol. 2, pp. 547–552, Springer (2023), DOI: 10.1007/978-3-031-30820-8_34
[16] Schwarz, M., Erhard, J., Vojdani, V., Saan, S., Seidl, H.: When long jumps fall short: Control-flow tracking and misuse detection for non-local jumps in C. In: SOAP '23, pp. 20–26, ACM (2023), DOI: 10.1145/3589250.3596140
[17] Schwarz, M., Saan, S., Seidl, H., Erhard, J., Vojdani, V.: Clustered relational thread-modular abstract interpretation with local traces. In: ESOP '23, pp. 28–58, Springer (2023), DOI: 10.1007/978-3-031-30044-8_2
[18] Seidl, H., Erhard, J., Schwarz, M., Tilscher, S.: 2-pointer logic. In: Javier Esparza's 60th Birthday, pp. 254–264, Springer (2024)
[19] Seidl, H., Vogler, R.: Three improvements to the top-down solver. Math. Struct. Comput. Sci. **31**(9), 1090–1134 (2021), DOI: 10.1017/S0960129521000499
[20] Sui, Y., Ye, D., Xue, J.: Static memory leak detection using full-sparse value-flow analysis. In: ISSTA '12, pp. 254–264, ACM (2012), DOI: 10.1145/2338965.2336784
[21] Wang, Z., Chen, Z.: AISE: A symbolic verifier by synergizing abstract interpretation and symbolic execution. In: TACAS '24, Springer (2024)