Symbiotic 9: String Analysis and Backward Symbolic Execution with Loop Folding*

(Competition Contribution)

Marek Chalupa✉, Vincent Mihalkovič, Anna Řechtáčková, Lukáš Zaoral, and Jan Strejček

Masaryk University, Brno, Czech Republic

Abstract. The development of Symbiotic 9 focused mainly on two components. One is the symbolic executor Slowbeast, which newly supports *backward symbolic execution* including its extension called *loop folding*. This technique can infer inductive invariants from backward symbolic execution states. Thanks to these invariants, Symbiotic 9 is able to produce non-trivial correctness witnesses, which is a feature that is missing in previous versions of Symbiotic. We have also extended forward symbolic execution in Slowbeast with a basic support for parallel programs. The second component with significant improvements is the instrumentation module. In particular, we have extended the static analysis of accesses to arrays with features designed for programs that manipulate C strings.
Symbiotic 9 is the *Overall* winner of SV-COMP 2022. Moreover, it won also the categories *MemSafety* and *SoftwareSystems*, and placed third in *FalsificationOverall*.

1 Verification Approach

Symbiotic 9 combines fast static analyses with code instrumentation and program slicing [13] to speed up the code verification. In the SV-COMP configuration of Symbiotic 9, the code verification is performed by symbolic executors, namely by Slowbeast [8] and our fork of Klee [4].

As Symbiotic works internally with llvm [10], it first compiles the given C program into llvm bitcode. The following steps depend on the verified property.

Verification of the Property `unreach-call` For this property, Symbiotic 9 directly slices the llvm bitcode to remove instructions that have no influence on the reachability of error calls and then run Klee with the time limit of 333 seconds. Klee is very efficient and often decides the task within this time limit. If Klee fails to decide, we parse its output and proceed according to the case of the failure. If Klee failed because the program contains threads, we

* This work has been supported by the Czech Science Foundation grant GA19-24397S.
✉ Jury member and the corresponding author: `chalupa@fi.muni.cz`

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 462–467, 2022.
https://doi.org/10.1007/978-3-030-99527-0_32

Table 1. The comparison of supported features of Klee (our fork and the upstream) and Slowbeast (SV-COMP 2022 and SV-COMP 2021 versions). The marks ✓/✓/✗ mean supported/partially supported/unsupported.

	Klee upstream	Klee our fork	Slowbeast SV-COMP 2021	Slowbeast SV-COMP 2022
Backward SE	✗	✗	✗	✓
Loop folding	✗	✗	✓	✓
Invariant generation	✗	✗	✓	✓
Symbolic floats	✗	✗	✓	✓
Symbolic pointers	✓	✓	✗	✓
Symbolic-sized allocations	✗	✓	✗	✓
Symbolic addresses	✗	✓	✗	✓
Parallel programs	✗	✗	✗	✓
Incremental solving	✗	✗	✓	✓
Caching solver calls	✓	✓	✗	✗
Lazy memory	✗	✗	✓	✓

run Slowbeast with forward *symbolic execution (SE)* and the threads support turned on. If Klee failed for any other reason, we run Slowbeast with *backward symbolic execution with loop folding (BSELF)* [8] described later. If BSELF also fails (the current implementation supports only selected program features), we run Slowbeast with forward symbolic execution.

Note that running forward symbolic execution first with Klee and then with Slowbeast if Klee fails does make a good sense as Klee and Slowbeast support a different set of features. The main differences between these tools (and the upstream Klee and the version of Slowbeast used in Symbiotic 8) are summarized in Table 1. Row *symbolic addresses* indicates whether tools model the non-determinism in the placement of allocated objects (this is useful, e.g., when comparing addresses of such objects). Row *incremental solving* indicates whether tools can associate the state of an SMT solver to every symbolic execution state and incrementally add constraints instead of always solving formulas from the scratch. Row *caching solver calls* indicates whether tools can remember results of solver calls and use them later to quickly decide some other solver calls. Finally, row *lazy memory* indicates if the tool can create memory objects on-demand when first accessing them, without their previous allocation (it assumes that the accesses to memory are valid). This feature is crucial when we want execute a program by parts, without starting from the entry point. The meaning of the remaining rows should be clear or is explained later.

If an error is found by either tool, it is replayed on the unsliced code. If the replay succeeds, we generate a violation witness. If no error is found and the analysis was complete, we generate a correctness witness. If the program correctness was proved by Slowbeast with BSELF, we generate a witness containing the computed invariants, otherwise we generate a trivial correctness witness as we have no invariants at hand. In all other cases, Symbiotic 9 answers *unknown*.

Verification of Other Properties For verification of other properties than `unreach-call`, Symbiotic 9 uses the same workflow as Symbiotic 8 [7]. In brief, the instrumentation module marks program instructions that can potentially violate the considered property. The module employs suitable fast static analyses to identify these instructions (e.g., when checking the property `no-overflow`, it uses a range analysis to discover the instructions that may perform a signed integer overflow). The bitcode with marked instructions is sliced such that the arguments and the reachability of these instructions are preserved. The sliced bitcode is passed to Klee. If it discovers a property violation and then replays it on the unsliced code, we produce a violation witness. If Klee completes its analysis without any property violation found, we produce a trivial correctness witness. In all other cases, Symbiotic 9 returns *unknown*.

Backward Symbolic Execution with Loop Folding (BSELF) [8] Slowbeast newly implements *backward symbolic execution (BSE)* [9], which explores the program backward from target locations towards the initial location and incrementally computes weakest preconditions for the explored program paths. BSE is a valuable technique on its own as it precisely corresponds to k-induction on control-flow paths [8]. *Loop folding* is a technique that aims to infer inductive invariants during BSE. Roughly speaking, when BSE starts from an error location and reaches a loop header, loop folding creates an initial *invariant candidate* that is disjoint with the current weakest precondition (i.e., the states that can reach the error location). If the invariant candidate is actually an invariant, we know that the error location is not reachable via the explored path. Otherwise, a pre-image of the invariant candidate along a loop path is computed, over-approximated, and added to the candidate. This process is repeated until an invariant is found or until it fails for some reason, e.g., when it discovers that the error location is actually reachable. Loop folding can infer complex disjunctive invariants and since it uses the error states, it is also property-driven.

String Analysis and Other Improvements The second major improvement in Symbiotic 9 is in the instrumentation for the property `valid-memsafety`. We have improved the analysis for the identification of out-of-bounds array accesses.

In Symbiotic 8, this analysis only determined whether an array access done via the index variable is in bounds [14]. The analysis in Symbiotic 9 also handles more general patterns where the array contains a concrete value (0 in the case of C strings) and the index pointer is incremented by one until it points to this concrete value, and where the pointer is incremented a fixed number of times.

Further, we have extended the forward symbolic execution in Slowbeast to handle parallel programs. For now, the symbolic execution is highly inefficient as it examines each interleaving of globally visible events. We plan to implement some reductions in the future. Slowbeast has been also extended to generate witnesses as this functionality was missing. Notably, it can generate non-trivial correctness witnesses using the invariants computed by BSELF. Previous versions of Symbiotic generate only trivial correctness witnesses.

Slicing has been also improved. It now applies a fast and coarse slicing before the main slicing. The coarse slicing detects all basic blocks from which no *slicing criterion* (i.e., an instruction whose reachability and arguments should be preserved) is syntactically reachable and replaces them by calls to `abort`.

2 Strengths and Weaknesses

Forward symbolic execution is unable to fully analyze unbounded loops or infinite execution paths. Hence, unless program slicing removes the unbounded computation from the program, forward symbolic execution cannot verify it. However, backward symbolic execution and BSELF can fully analyze at least some unbounded programs [8]. Still, both these methods are computationally complex as the number of paths they must search may be enormous and their exploration may involve many non-trivial calls to the SMT solver. Therefore, these methods do not scale to real-world programs.

A strong aspect of Symbiotic is the very interplay of fast static analyses in the instrumentation, program slicing, and forward and backward symbolic execution. Fast static analyses are able to deem correct many parts of the code (with respect to the verified property). These parts of the code are then usually removed by slicing and only the possibly unsafe parts of the program (and their dependencies) get into a symbolic executor. In this sense, Symbiotic does incremental or conditional [3] verification.

Results of Symbiotic 9 in SV-COMP 2022 In SV-COMP 2022 [1], Symbiotic 9 won categories *MemSafety*, *SoftwareSystems*, and *Overall*, and got the 3rd place in *FalsificationOverall*. Moreover, it produced 1529 correct answers that were not confirmed, which is the highest number in SV-COMP 2022. 1073 unconfirmed answers are in *MemSafety-Juliet*, where we produced some incorrect witnesses due to a bug. Another 258 unconfirmed answers are in *Termination*. Symbiotic 9 produced only 3 incorrect answers caused by a bug in the replay mode of Slowbeast.

3 Software Project and Contributors

All components of Symbiotic 9 use llvm 10 [10]. Slicer and instrumentation module are written in C++ and extensively use the library DG [5]. Klee is implemented in C++ and Slowbeast [12] is written in Python. Both symbolic executors use Z3 [11] as the SMT solver. Control scripts are written in Python.

Symbiotic 9 and all its components and external libraries are available under open-source licenses that comply with SV-COMP's policy for the reproduction of results. Symbiotic 9 participated in all categories of SV-COMP 2022 except the categories with Java programs.

Symbiotic 9 has been developed by Marek Chalupa, Vincent Mihalkovič, Anna Řechtáčková, and Lukáš Zaoral under the supervision of Jan Strejček.

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 Symbiotic used in the competition is archived together with other participating tools [2] and also in its own artifact [6] at Zenodo.

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., Jakobs, M.: Fred: Conditional model checking via reducers and folders. In: SEFM 2020. LNCS, vol. 12310, pp. 113–132. Springer (2020). https://doi.org/10.1007/978-3-030-58768-0_7
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
5. Chalupa, M.: DG: analysis and slicing of LLVM bitcode. In: ATVA 2020. LNCS, vol. 12302, pp. 557–563. Springer (2020), https://doi.org/10.1007/978-3-030-59152-6_33
6. Chalupa, M.: Symbiotic 9: String analysis and backward symbolic execution with loop folding (artifact). Zenodo (2022). https://doi.org/10.5281/zenodo.5947909
7. Chalupa, M., Jašek, T., Novák, J., Řechtáčková, A., Šoková, V., Strejček, J.: Symbiotic 8: Beyond symbolic execution - (competition contribution). In: TACAS 2021. LNCS, vol. 12652, pp. 453–457. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_31
8. Chalupa, M., Strejček, J.: Backward symbolic execution with loop folding. In: SAS 2021. LNCS, vol. 12913, pp. 49–76. Springer (2021). https://doi.org/10.1007/978-3-030-88806-0_3
9. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In: PLDI 2009. pp. 363–374. ACM (2009). https://doi.org/10.1145/1542476.1542517
10. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004. pp. 75–88. IEEE Computer Society (2004), https://doi.org/10.1109/CGO.2004.1281665
11. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24
12. Slowbeast repository. https://gitlab.com/mchalupa/slowbeast (2021)
13. Weiser, M.: Program slicing. In: Proceedings of ICSE. pp. 439–449. IEEE (1981)
14. Řechtáčková, A.: Improving out-of-bound access checking in Symbiotic (2020), https://is.muni.cz/th/tmq7m/, bachelor thesis, accessed 2022-02-02