Symbiotic 10: Lazy Memory Initialization and Compact Symbolic Execution*
(Competition Contribution)

Martin Jonáš[1](✉), Kristián Kumor[1], Jakub Novák[1], Jindřich Sedláček[1], Marek Trtík[1], Lukáš Zaoral[2], Paulína Ayaziová[1], and Jan Strejček[1]

[1] Masaryk University, Brno, Czech Republic
`martin.jonas@mail.muni.cz`
[2] Red Hat, Brno, Czech Republic

Abstract. Symbiotic 10 brings four substantial improvements. First, we extended our clone of Klee called JetKlee with *lazy memory initialization*. With this extension, JetKlee can symbolically execute a function without knowing its context. In SV-COMP, we use it to handle `extern` variables. Second, we have implemented the technique called *compact symbolic execution* to Slowbeast. Third, we have implemented a non-trivial *may-happen-in-parallel* analysis, which improves slicing of parallel programs. Finally, we have implemented support for violation witnesses in the new *witness format 2.0*.

1 Verification Approach

Just like previous versions, Symbiotic 10 relies on a combination of static analysis, code instrumentation, and several flavors of *symbolic execution* (SE) [8]. It employs two symbolic executors: Slowbeast and our fork of Klee [2] called JetKlee. Slowbeast implements standard (forward) SE, backward SE with loop folding [5], and compact SE [13]. JetKlee implements standard SE.

The rest of the section describes the precise workflow for various types of properties and discusses the differences between Symbiotic 10 and Symbiotic 9.1, which is the version that competed in SV-COMP 2023.

Verification of the Property `unreach-call` For this property, Symbiotic 10 performs slicing of the given program to remove the parts that have no influence on reaching the target function, and executes sequential portfolio of the following engines. Each of the engines is executed for the given number of seconds. The execution can be shorter if the engine decides or fails to decide, e.g., due to an unsupported feature of the input program like threads or symbolic floats.

1. Forward symbolic execution by JetKlee for 333 seconds. JetKlee is efficient industrial-strength symbolic executor and most of the solved benchmarks are solved by JetKlee.

* This work has been supported by the Czech Science Foundation grant GA23-06506S.
✉ Jury member

2. Compact symbolic execution (CSE) [13] by Slowbeast for 60 seconds. In most cases, CSE either finishes quickly or brings no benefit compared to standard forward symbolic execution.
3. Backward symbolic execution with loop folding (BSELF) [5] by Slowbeast without time limit.
4. If BSELF fails, we perform forward symbolic execution by Slowbeast without time limit. The reason for this is that Slowbeast has better support for floating point arithmetic and threads than JetKlee.

If an error is found by any of the engines, it is replayed on the unsliced code. If the replay succeeds, we generate a violation witness. If the program is decided safe by BSELF, we generate a correctness witness containing the generated invariants. The other engines do not support invariant generation, therefore if the program is decided safe by any other of the engines, we generate a trivial correctness witness.

Verification of Other Properties For other properties, Symbiotic 10 uses the same workflow as Symbiotic 9 [4]. In a nutshell, we identify program instructions that can violate the property, instrument the program with code that dynamically checks the property violation before each of the identified instructions, slice the program, and run either JetKlee or Slowbeast.

Compact Symbolic Execution We extended Slowbeast with *compact symbolic execution* (CSE) [13]. CSE analyzes each looping path of the execution and tries to summarize it by a quantified formula that describes the effect of κ iterations of that cyclic path, where κ is a free variable. For example, if we apply compact symbolic execution to the loop

```
while (i < n) { if (A[i] = 0) { break; }; i += 2; },
```

the path condition will be augmented by the quantified formula

$$\kappa \geq 0 \quad \wedge \quad \forall \tau. \; (0 \leq \tau < \kappa \rightarrow (i + 2\tau < n \wedge A[i + 2\tau] \neq 0))\,.$$

This allows symbolic execution to fully explore some programs with unbounded loops and find deep counterexamples. However, it works only for looping paths of specific form and requires potentially expensive quantified SMT reasoning.

Lazy Memory Initialization We extended JetKlee with *lazy memory initialization*, which constructs symbolic memory objects lazily during the first access to that object, not during its initialization. This allows isolated symbolic execution of functions without knowing their arguments and calling context. As all programs in SV-COMP start with the `main` function and there is no need to analyze an isolated function, we use this feature in the competition only to support externally defined variables. Note that this cannot be achieved by merely making the externally defined variable symbolic, as it can be a pointer to external memory, which needs to be properly initialized. For this reason, externally defined variables were not supported by the previous version of Symbiotic.

Table 1. The comparison of Symbiotic 9.1 and Symbiotic 10 on the intersection of benchmarks from SV-COMP 2023 and SV-COMP 2024. The table is computed from the official results of SV-COMP 2023 and SV-COMP 2024.

Property	Benchmarks	Both solved	Only 10 solved	Only 9.1 solved
`no-data-race`	783	0	0	0
`no-overflow`	7502	442	4102	1
`termination`	1809	1220	10	31
`unreach-call`	9537	3577	116	225
`valid-memcleanup`	61	35	0	0
`valid-memsafety`	4113	416	1427	34

May-Happen-in-Parallel Analysis We improved slicing of parallel programs by employing a static *may-happen-in-parallel* analysis [11], which overapproximates the set of pairs of program locations that can happen in parallel in different threads. Previously, Symbiotic assumed that all possible pairs of instructions can happen in parallel, which reduced effectivity of slicing. The implementation currently does not consider thread synchronization. For more details, see the bachelor's thesis about the implementation [12]. In the future, we want to use this analysis also for proving some `no-data-race` properties.

Other Changes All external dependencies of Symbiotic 10 have been updated to newer versions and all parts of Symbiotic 10 have been ported to llvm 14. Notably, this concerns JetKlee, into which we merged most of the upstream changes from the base Klee (more than 300 commits).

We extended JetKlee with support for generating yaml-based violation witnesses in witness format 2.0[3]. Slowbeast still supports only the older witness format 1.0 based on GraphML.

We also fixed incorrect overflow checking of 64-bit integers and incorrect modeling of `fscanf` for the purposes of static analysis and instrumentation. Due to these problems, Symbiotic 9.1 did not support any of `*-Juliet` benchmarks, which are now fully supported.

Unlike the previous versions of Symbiotic, Symbiotic 10 does not employ Predator [6] as a static analyzer. This is due to technical difficulties during porting our version of Predator to llvm 14. This is a temporary solution and we plan include Predator in the future versions of Symbiotic.

2 Strengths and Weaknesses

Standard forward symbolic execution suffers from path explosion and is unable to fully analyze programs with unbounded loops. Backward symbolic execution with loop folding and compact symbolic execution can finish analysis even for

[3] https://gitlab.com/sosy-lab/benchmarking/sv-witnesses

some programs with unbounded loops, yet they still suffer from path explosion and will time out on programs with a large number of branching paths.

The results of SV-COMP 2024 show that the combination of static analysis, instrumentation, program slicing, and several variants of symbolic execution are efficient in practice, in particular for bug hunting. The static analyses are often able to prove that some parts of the code are correct or do not influence the property. These parts of the code then can be removed by slicing. This partly mitigates the scalability problem caused by path explosion.

Results of Symbiotic 10 in SV-COMP 2024 Symbiotic 10 participated in all categories of SV-COMP 2024 for C programs. It won silver medals in categories *MemSafety* and *FalsificationOverall* [1]. Symbiotic 10 produced 19 wrong answers; most of these are caused by imprecise modeling of the system functions `setlocale` and `getopt_long`. They are not fundamental problems of the approach and will be fixed.

Table 1 compares the results of Symbiotic 9.1 in SV-COMP 2023 and Symbiotic 10 in SV-COMP 2024 on the benchmarks that were used in both years. Symbiotic 10 was able to correctly solve 5655 benchmarks that were not solved by Symbiotic 9.1. From these, 5366 benchmarks (3990 `no-overflow` + 1376 `valid-memsafety`) are from subcategories `*-Juliet`, which the previous version of Symbiotic did not support. Unfortunately, 147 of the previously decided benchmarks from `ConcurrencySafety-main` with property `unreach-call` were not decided by Symbiotic 10 due to a bug in our version of Slowbeast. Additionally, 31 of previously decided benchmarks (16 in `Memsafety-Heap` and 15 in `Memsafety-LinkedLists`) were not decided by Symbiotic 10 due to exclusion of Predator. If Predator had not been excluded or the wrong results had been fixed, Symbiotic 10 would have won the *MemSafety* category.

3 Software Architecture, Usage, and Contributors

All components of Symbiotic 10 use llvm 14 [9] for the intermediate representation. To obtain the llvm bitcode from the verified C program, Symbiotic relies on clang. Slicer and instrumentation module are written in C++ and rely on the library DG [3]. JetKlee is implemented in C++ and Slowbeast [14] is written in Python. Both symbolic executors use Z3 [10] as the smt solver. Control scripts are written in Python. All the components and external dependencies have permissive open-source licenses.

Binary form of Symbiotic 10 is available Zenodo [7], source code is available from https://github.com/staticafi/symbiotic under the tag `svcomp24`. You can run Symbiotic with

```
bin/symbiotic --sv-comp --prp <prpfile> [--32] <source>.
```

For details, see the file `README.md` in the mentioned repository.

Symbiotic 10 has been developed at the Faculty of Informatics of Masaryk University by the authors of this paper under the supervision of Jan Strejček.

References

1. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
2. 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
3. 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
4. Chalupa, M., Mihalkovič, V., Řechtáčková, A., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding - (Competition Contribution). In: Fisman, D., Rosu, G. (eds.) TACAS 2022. Lecture Notes in Computer Science, vol. 13244, pp. 462–467. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_32, https://doi.org/10.1007/978-3-030-99527-0_32
5. 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, https://doi.org/10.1007/978-3-030-88806-0_3
6. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer (2011), https://doi.org/10.1007/978-3-642-36742-7_49
7. Jonáš, M., Kumor, K., Novák, J., Sedláček, J., Shandilya, S., Trtík, M., Zaoral, L., Strejček, J.: Symbiotic 10: Submission to SV-COMP 2024 (Nov 2023). https://doi.org/10.5281/zenodo.10202594
8. King, J.C.: Symbolic execution and program testing. Communications of ACM **19**(7), 385–394 (1976)
9. 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
10. 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
11. Naumovich, G., Avrunin, G.S., Clarke, L.A.: An Efficient Algorithm for Computing *MHP* Information for Concurrent Java Programs. In: Nierstrasz, O., Lemoine, M. (eds.) ESEC / SIGSOFT FSE 1999. Lecture Notes in Computer Science, vol. 1687, pp. 338–354. Springer (1999). https://doi.org/10.1007/3-540-48166-4_21, https://doi.org/10.1007/3-540-48166-4_21
12. Sedláček, J.: May-Happen-in-Parallel Analysis for Slicing of Parallel Programs. Bachelor's thesis, Masaryk University (2024), https://is.muni.cz/th/he6cd/
13. Slaby, J., Strejček, J., Trtík, M.: Compact symbolic execution. In: Hung, D.V., Ogawa, M. (eds.) ATVA 2013. Lecture Notes in Computer Science, vol. 8172, pp. 193–207. Springer (2013). https://doi.org/10.1007/978-3-319-02444-8_15, https://doi.org/10.1007/978-3-319-02444-8_15
14. Slowbeast repository. https://gitlab.com/mchalupa/slowbeast (2021)