






Symbiotic 8: Beyond Symbolic Execution*

(Competition Contribution)

Marek Chalupa¹, Tomáš Jašek¹, Jakub Novák¹,
Anna Řečtáčková¹, Veronika Šoková², and
Jan Strejček¹



¹ Masaryk University, Brno, Czech Republic

² Brno University of Technology, FIT, Brno, Czech Republic

Abstract. SYMBIOTIC 8 extends the traditional combination of static analyses, instrumentation, program slicing, and symbolic execution with one substantial novelty, namely a technique mixing symbolic execution with k-induction. This technique can prove the correctness of programs with possibly unbounded loops, which cannot be done by classic symbolic execution. SYMBIOTIC 8 delivers also several other improvements. In particular, we have modified our fork of the symbolic executor KLEE to support the comparison of symbolic pointers. Further, we have tuned the shape analysis tool PREDATOR (integrated already in SYMBIOTIC 7) to perform better on LLVM bitcode. We have also developed a light-weight analysis of relations between variables that can prove the absence of out-of-bound accesses to arrays.

1 Verification Approach

SYMBIOTIC is a program analysis framework that combines fast static analyses with code instrumentation and program slicing to speed up the code verification which is then performed by symbolic executor KLEE [3] (or, alternatively, by another supported verification tool). The main improvement in SYMBIOTIC 8 is a new verification technique combining symbolic execution with k-induction [8] that we call *KindSE*.

Symbolic execution with k-induction (KindSE) *KindSE* applies the idea of k-induction [8] to paths of the control flow graph. The approach can be roughly described by the following three steps.

1. Set k to 1. Let P be the set of all paths in the control flow graph of length k that end in an error location.
2. Use symbolic execution to execute every path $\pi \in P$. If the symbolic execution says that π is infeasible, remove π from P . If π is feasible and it starts in the initial location, report that the program is incorrect.

* This work has been supported by the Czech Science Foundation grant GA20-07487S.

✉ Jury member and the corresponding author: chalupa@fi.muni.cz.

3. If P is empty, the control flow graph contains no feasible path of length k (or more) leading to an error location and thus we report that the program is correct. If P is not empty, we replace each path $\pi \in P$ by paths of length $k + 1$ that have π as its suffix, increase k by one, and go to step 2.

To improve the performance, we further extended the algorithm to summarize loop iterations. If we process a program location that is a loop header, we start unwinding the loop backwards. We over-approximate the states that we get in every loop iteration to cover more than one iteration if possible. If we are successful, the summarized loop states form an inductive invariant, which can help to prove that no error location is reachable from the loop header in k steps. Our loop summarization does not handle nested loops (in this case we fall-back to the algorithm without loop summarization) and calls of functions. To fix the latter restriction, we inline all procedures (if possible) before running KindSE.

KindSE is implemented in our prototype tool SLOWBEAST [1] which we integrated into SYMBIOTIC 8. The tool now supports only the `unreach-call` property. SLOWBEAST can also work as a standard symbolic executor (without k-induction), but it is noticeably slower than KLEE and it has some limitations. However, it supports symbolic floating point arithmetics, which KLEE does not.

Workflow of Symbiotic 8 As the first step, a given program is translated to LLVM [6]. If the program contains a call to `pthread.create`, SYMBIOTIC returns `unknown` as it cannot handle parallel programs. The rest of the workflow then depends on the verified property, as indicated in Figure 1.

For `unreach-call` property, we call slicer to remove instructions that have no influence on the property and run KLEE. If KLEE does not decide in 222 seconds, we run KindSE in SLOWBEAST. If it fails, we run KLEE again and if it also fails, we run SLOWBEAST as a standard symbolic executor. If some tool says

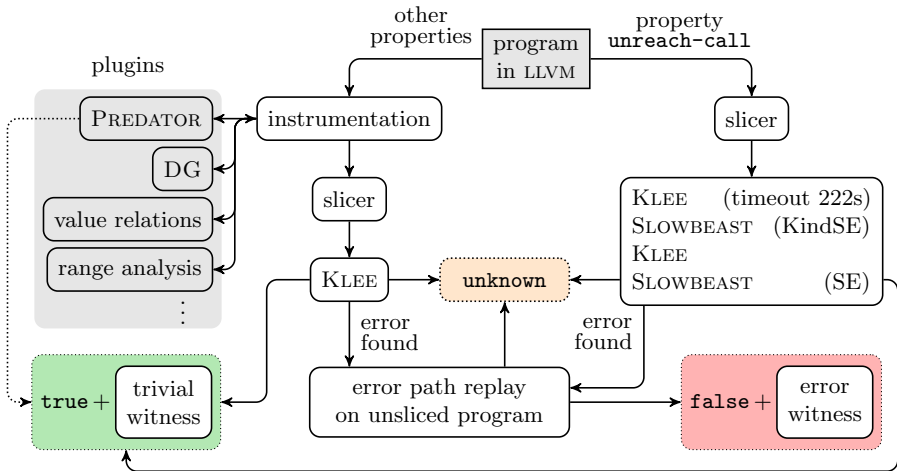


Fig. 1. The workflow of SYMBIOTIC 8

that the specified call is unreachable, we return `true` with the trivial witness. If we detect that the specified call is reachable, we try replaying the error path on the unsliced program. If the replay confirms that the call is reachable, we return `false` with the error witness generated from the replay.

For other properties, we instrument the program with the help of various analyses. For example, when checking memory safety, we use PREDATOR [5], DG [4], and a values-relations analysis to detect potentially unsafe instructions. If PREDATOR says that all instructions are safe, we directly return `true`. Otherwise, we slice the program with respect to potentially unsafe instructions and call KLEE. The rest of the process is identical to the previous case.

2 Software Architecture

All components of SYMBIOTIC 8 use LLVM 10 [6]. Scripts that call and control the components according to a given configuration are written in Python.

Instrumentation module is written in C++. In SYMBIOTIC 8, we have newly integrated a values-relations analysis as a plugin into instrumentation. This analysis is able to prove valid some accesses into arrays. We have also improved LLVM frontend of PREDATOR [5] to perform similarly well as the GCC frontend.

Program slicing module is written in C++ and is build around the library DG [4]. This year, we sped up the slicer by using more efficient data structures in pointer analysis and by using function summaries in data dependence analysis.

We use our own fork of KLEE [3] that differs from the upstream KLEE mainly in using segment-offset pointer representation which allows for better handling of symbolic pointers and symbolic-sized allocations. This year, we mended handling of symbolic pointers and added support for comparison of symbolic addresses.

Tool SLOWBEAST [1] is written in Python. Both, KLEE and SLOWBEAST use Z3 [7] as the SMT solver.

3 Strengths and Weaknesses

Symbolic execution may be very efficient in finding bugs but suffers from the *path explosion problem* which may prevent it from fully analyzing programs with high level of branching. We alleviate this problem by using program slicing. However, in the presence of unbounded loops or infinite execution paths, program slicing does not help unless it removes the unbounded computation from the program. Indeed, classical symbolic execution is unable to verify such programs at all.

To fight the inability of symbolic execution to verify unbounded programs, we use *KindSE*. However, its implementation in SLOWBEAST is still not fully matured and it handles only a very restricted set of programs.

Results of Symbiotic 8 in SV-COMP 2021 SYMBIOTIC 8 won *MemSafety* and *SoftwareSystems* categories [2]. In the *MemSafety* category, we lost many points in the new *MemSafety-Juliet* subcategory. These benchmarks contain

threads and SYMBIOTIC immediately answered *unknown* due to the syntactic check mentioned in Section 1. However, most of these benchmarks actually do not spawn any thread and thus SYMBIOTIC could analyze them. The victory in *SoftwareSystems* category is mainly due to the dominance on the new *uthash* benchmarks.

This year, over 500 correct answers produced by SYMBIOTIC were not confirmed. Some of these cases must be accounted to the fact that SYMBIOTIC generates only trivial correctness witnesses. However, there are also unconfirmed answers because of missing witnesses, which turned out to be a bug in SLOW-BEAST integration. Unfortunately, these include all 99 benchmarks that were newly proved correct by KindSE, from which 85 were in the *ReachSafety-Loops* subcategory. We had also many unconfirmed witnesses for non-termination violation that still need to be investigated.

SYMBIOTIC had 16 incorrect answers: 14 incorrect *true* in *Termination* category and 2 incorrect *false* in *ReachSafety-Floats*. All of them were caused by last-minute commits that were fixed shortly after the submission deadline. Because of these mistakes, SYMBIOTIC ended up on the 4th place instead of on the 2nd in the *Termination* category.

In the *Overall* meta-category, SYMBIOTIC traditionally took the 4th place as every year since 2018.

4 Tool Setup and Project Contributors

The archive is available at <https://doi.org/10.5281/zenodo.4483882>. Run SYMBIOTIC as:

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

The option `--prp` sets the verified property and `--32` tells SYMBIOTIC to assume 32-bit architecture (64-bit architecture is assumed by default).

5 Software Project and Contributors

SYMBIOTIC 8 for SV-COMP 2021 has been developed by Marek Chalupa, Tomáš Jašek, Jan Novák, and Anna Řečtáčková under the supervision of Jan Strejček. Veronika Šoková provided a valuable help with adjusting PREDATOR modifications. SYMBIOTIC is available under the MIT license. All the external components that the tool uses are also available under open-source licenses that comply with SV-COMP's policy for the reproduction of results. The source code of SYMBIOTIC can be found at:

<https://github.com/staticafi/symbiotic>

References

1. SLOWBEAST. <https://gitlab.fi.muni.cz/xchalup4/slowbeast/> (2020)
2. Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: TACAS 2021. LNCS 12652, Springer (2021)
3. 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
4. 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
5. 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
6. 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>
7. 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
8. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer (2000), https://doi.org/10.1007/3-540-40922-X_8

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

