






SYMBIOTIC-WITCH 2: More Efficient Algorithm and Witness Refutation[★] (Competition Contribution)

Paulína Ayaziová  and Jan Strejček  

Masaryk University, Brno, Czech Republic
{xayaziov, strejcek}@fi.muni.cz

Abstract. The new version of the witness validator SYMBIOTIC-WITCH follows more precisely the (fixed version of the) semantics of verification witnesses. This makes the tool more efficient as it can benefit from sink nodes. Further, the tool can now refute a witness. To sum up, SYMBIOTIC-WITCH 2 can confirm or refute violation witnesses of *reachability safety*, *memory safety*, *memory cleanup*, and *overflow* properties of sequential C programs.

1 Witness Validation Approach

The basic principle of the witness validator SYMBIOTIC-WITCH 2 remains the same as in the previous version of the tool [1], i.e., it symbolically executes [9] the given program along execution paths specified by the corresponding witness. The substantial differences were induced by a more precise interpretation of violation witnesses and by the community decision to support witness refutation.

We originally thought that every node of a witness automaton has an implicit self-loop that can be taken under each program instruction. After SV-COMP 2022, we learnt that the implicit self-loop of a node q can be used only by edges of *control flow automata (CFA)* that are “either

- (a) not matched by the source-code guard of any other outgoing transition of q
or
- (b) are matched by the source-code guard of some other outgoing transition of q that also matches a successor CFA edge.” [5]

This definition is problematic in particular because it refers to CFA and there is no standardized translation of C programs to CFA. Especially the case (b) heavily depends on the granularity of constructed CFA as it refers to adjacent edges. As the semantics of verification witnesses has to be unambiguous, we have convinced the community that the case (b) should be removed from the semantics. Still, the case (a) is viable and it considerably reduces the applicability of implicit self-loops.

[★] This work has been supported by the Czech Science Foundation grant GA23-06506S.

SYMBIOTIC-WITCH 2 works as follows. It reads a given violation witness and the corresponding program. The program is symbolically executed and every state of symbolic execution is accompanied by the set of witness automaton nodes that are reached by the executed program path. Note that these sets are dramatically smaller than in the previous version of our tool due to the more precise semantics of implicit self-loops. If the set does not contain any node except sink nodes, the symbolic execution of the corresponding path is stopped. This brings a significant speed up compared to the previous version of our tool where this situation cannot happen.

Another significant difference to the previous version is the handling of state-space guards of a given witness. Consider a symbolic execution state and the associated set of witness automata nodes. Further, assume that the next instruction processed by the symbolic execution matches the source-code guards of some automata edges leading from the set of nodes. For each state-space guard of these edges, we create a fork of symbolic execution and restrict the next symbolic execution state to satisfy the state-space guard. The set of nodes accompanying the restricted symbolic execution state contains only target nodes of the edges with the enforced state-space guard. Note that the previous version of our validator ignores state-space guards unless the witness automaton contains a single path from the entry node to the violation node.

If the symbolic execution detects a violation of the considered property and the tracked set of witness automata nodes contains a violation node, the witness is confirmed. The witness is refuted if

- the symbolic execution ends without finding a property violation represented by the witness and
- there was no execution path unexplored due to the limitations of the employed symbolic executor (e.g., our executor based on KLEE [6] cannot handle symbolic floats and thus it instantiates them with a concrete value and ignores executions with other values) and
- the witness uses only source-code guards supported by our tool (see below).

The witness automata use various attributes to specify source-code guards (saying which instructions correspond to a given witness automaton edge) and state-space guards (restrictions on program states). SYMBIOTIC-WITCH 2 supports only selected attributes for source-code guards, namely the line number of executed instructions, the information whether *true* or *false* branch is taken, and the information about entering a function or returning from a function. Regarding the state-space guard, our tool uses only the return values of the `__VERIFIER_nondet_*` functions. The limited support of attributes means that our tool can misinterpret a given witness automaton, i.e., it can consider some execution path to be represented by the automaton even if it is not, and vice versa. In practice, this is not a big issue as many verification tools produce violation witnesses with only the supported attributes and some other tools use unsupported attributes to provide additional information (like offset of an instruction in the source code) that typically do not change the represented set of execution paths.

2 Software Architecture

The tool SYMBIOTIC-WITCH 2 is integrated to the SYMBIOTIC framework [7] and it can be roughly divided into two components. The first component is a set of python scripts (many of them shared with other SYMBIOTIC tools) that preprocess the code. More precisely, they set the options for optimisations and CLANG sanitizer depending on the considered property, translates the given C program into LLVM intermediate representation via CLANG, and links necessary function definitions.

The second component called WITCH-KLEE takes the preprocessed program and the witness, and it runs the actual witness validation. WITCH-KLEE is derived from the symbolic executor JETKLEE, which is a fork of KLEE [6] used in the SYMBIOTIC framework. WITCH-KLEE employs RAPIDXML for parsing witnesses in the GraphML format [5] and Z3 [10] as the SMT solver in symbolic execution.

Both components of SYMBIOTIC-WITCH 2 run on LLVM 10.0.1.

3 Strengths and Weaknesses

On the positive side, SYMBIOTIC-WITCH 2 can efficiently handle violation witnesses providing return values of `__VERIFIER_nondet_*` functions as well as those describing execution paths by taken branches.

Further, if SYMBIOTIC-WITCH 2 confirms a witness containing only attributes supported by the tool, then the witness is indeed valid. If SYMBIOTIC-WITCH 2 confirms a witness with some attributes not supported by the tool, then the program really violates the considered property and this violation can, but does not have to be represented by the witness. If SYMBIOTIC-WITCH 2 refutes a witness, then this witness is indeed invalid. The only exception is the case when the program contains some inner nondeterminism that is lost by the translation to LLVM. For example, consider a program that contains a test $f(\mathbf{x}) < g(\mathbf{x})$. Due to the C standard, the functions $f(\mathbf{x})$ and $g(\mathbf{x})$ can be evaluated in any order. If a violation witness prescribes one order of evaluation and CLANG translates the program such that the functions are evaluated in the opposite order, then the witness can be refuted even if it is correct. We can construct such a witness, but we have not yet come across any of these in practice. We plan to extend our tool with a check for this kind of inner nondeterminism in order to guarantee the correctness of refutation answers.

Our tool also has some weaknesses. Some of them come from the fact that we do not support all possible attributes of witnesses. We decided not to invest more effort to support other attributes as we expect the witness format to be revised soon due to detected issues in its semantics. In spite of this, the tool correctly confirmed 35536 and refuted 3108 violation witnesses of SV-COMP 2023. On the negative side, the tool also confirmed 10 witnesses of memory safety violation marked as invalid. Nine of these incorrect validation results stem from two verification tasks where our symbolic executor reported a `valid-memtrack` violation while the tasks are marked *true* for this property.

SYMBIOTIC-WITCH 2 struggles to evaluate two specific classes of witnesses. The first class are the witnesses for the programs in the ECA subcategory. These generated artificial programs are hard to compile and optimize. Thus, our tool sometimes runs out of time during the code preprocessing phase.

The second class are the witnesses that contain edges describing declarations and initializations of global variables (e.g., some witnesses produced by Ultimate Automizer [8]). Our algorithm processes these declarations and initializations in a separate step and starts the symbolic execution of a given program (and thus also the witness tracking) in the function `main`. This means that the witness tracking cannot pass any witness edge representing instructions that are not reachable from `main`. Hence, SYMBIOTIC-WITCH 2 can refute some witnesses of the second class even if it finds the property violations they represent. This issue can be seen as another consequence of the fact that the semantics of witnesses is formulated over CFA and the translation of C programs to CFA is not given.

4 Tool Setup and Configuration

The archive with SYMBIOTIC-WITCH 2 is available in the SV-COMP archives. To run the validator, use the command

```
./symbiotic [--prp <prop>] [--32 | --64] --witness-check <witness> <prg>
```

where `<witness>` is a violation witness in the GraphML format, `<prg>` is the corresponding C program, and `<prop>` is the considered property. The property can be supplied as a `.prp` file or one of the following shortcuts: `no-overflow`, `valid-memsafety`, or `valid-memcleanup`. The default property is unreachability of the function `reach_error()`. The switches `--32` and `--64` specify the considered architecture, 64-bit being the default.

Both components of the tool are also available on GitHub with build instructions in the respective `README.md` files. To start validation, build each component separately, add the path to the built `witch-klee` executable to `$PATH` and run SYMBIOTIC as previously described.

5 Software Project and Contributors

SYMBIOTIC-WITCH 2 has been developed at Faculty of Informatics, Masaryk University by Paulína Ayaziová under the guidance of Jan Strejček. The tool is available under the MIT license and all used tools and libraries (LLVM, KLEE, Z3, RAPIDXML, SYMBIOTIC) are also available under open-source licenses that comply with SV-COMP's policy for the reproduction of results. The source code of WITCH-KLEE (the competing version tagged SV-COMP23) can be found at:

<https://github.com/ayazip/witch-klee>

The source code of the respective version of SYMBIOTIC is available at:

<https://github.com/staticafi/symbiotic/tree/witch-klee>

Data Availability Statement. All data of SV-COMP 2023 are archived as described in the competition report [3] and available on the [competition web site](#). This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of SYMBIOTIC-WITCH 2 used in the competition is archived together with other participating tools [4] or separately [2].

References

1. Ayaziová, P., Chalupa, M., Strejček, J.: Symbiotic-Witch: A Klee-based violation witness checker (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 468–473. Springer (2022), https://doi.org/10.1007/978-3-030-99527-0_33
2. Ayaziová, P., Strejček, J.: Symbiotic-Witch 2. Zenodo (2023). <https://doi.org/10.5281/zenodo.7630406>
3. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). LNCS, Springer (2023)
4. Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). <https://doi.org/10.5281/zenodo.7627829>
5. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Eng. Methodol. **31**(4), 57:1–57:69 (2022). <https://doi.org/10.1145/3477579>, <https://doi.org/10.1145/3477579>
6. 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
7. Chalupa, M., Mihalkovič, V., Řečtářková, A., Zaoral, L., Strejček, J.: Symbiotic 9: String analysis and backward symbolic execution with loop folding (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13244, pp. 462–467. Springer (2022), https://doi.org/10.1007/978-3-030-99527-0_32
8. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Hoenicke, J., Li, Y., Nutz, A., Musa, B., Schilling, C., Schindler, T., Podelski, A.: Ultimate Automizer and the search for perfect interpolants - (competition contribution). In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10806, pp. 447–451. Springer (2018), https://doi.org/10.1007/978-3-319-89963-3_30
9. King, J.C.: Symbolic execution and program testing. Communications of ACM **19**(7), 385–394 (1976), <https://doi.org/10.1145/360248.360252>
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

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.

