Symbiotic-Witch: A Klee-Based Violation Witness Checker[⋆] (Competition Contribution)

Paulína Ayaziová, Marek Chalupa, and Jan Strejček

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{xayaziov,chalupa,strejcek}@fi.muni.cz

Abstract. Symbiotic-Witch is a new tool for checking violation witnesses in the GraphML-based format used at SV-COMP since 2015. Roughly speaking, Symbiotic-Witch symbolically executes a given program with Klee and simultaneously tracks the set of nodes the witness automaton can be in. Moreover, it reads the return values of nondeterministic functions specified in the witness and uses them to prune the symbolic execution. The violation witness is confirmed if the symbolic execution reaches an error and the current set of witness nodes contains a matching violation node.
Symbiotic-Witch currently supports violation witnesses of *reachability safety*, *memory safety*, *memory cleanup*, and *overflow* properties.

1 Verification Approach

We present a new checker of violation witnesses called Symbiotic-Witch. The checker first loads a given violation witness in the GraphML format [5] and a given program. Then it performs symbolic execution [11] of the program and simultaneously tracks the progress of the execution in the witness automaton. More precisely, every state of symbolic execution is accompanied by the set of witness automaton nodes that can be reached under the executed program path. 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.

Note that the original description of the witness format [5] does not provide any formal semantics of the format. We interpret it in the way that if an edge in a witness automaton matches an executed program instructions, then we can follow the edge but we can also stay in its starting node. Hence, if we have the set of witness automaton nodes reached under a certain program path, then prolongation of this path can add some nodes to this set, but it never removes any node from the set. A brief reading of an upcoming detailed description of the format [4] reveals that it can be the case that an edge matching an executed program instruction has to be taken. If this is indeed the case, we will adjust

⋆ This work has been supported by the Czech Science Foundation grant GA19-24397S.

our tool, but the current implementation and the following texts consider the former semantic.

Before Symbiotic-Witch starts the symbolic execution, we remove from the witness automaton all nodes that are not on any path from the entry node to a violation node. In general, witness automata are related to program executions using node and edge attributes. Symbiotic-Witch currently supports only some attributes of witness edges to map a program execution to a given witness automaton. Namely, it uses 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. Additionally, if the witness automaton contains a single path from the entry node to a violation node and there is some information about return values of the `__VERIFIER_nondet_*` functions on this path, then we use these values in the symbolic execution of the program. Return values not provided in the witness are treated as symbolic values.

A more precise description of the approach can be found in the bachelor's thesis of P. Ayaziová [1].

2 Software Architecture

The approach has been implemented in a tool called Symbiotic-Witch, which is basically a modification of the symbolic executor Klee [8]. More precisely, it is derived from the clone of Klee used in Symbiotic, which employs the SMT solver Z3 [13] and supports symbolic pointers, memory blocks of symbolic sizes etc. For parsing of witnesses in the GraphML format, we use the library RapidXML.

As Klee executes programs in llvm [12], a given C program has to be translated to llvm first. We use Clang for this translation as explained in Section 4.

The current version of Symbiotic-Witch runs on llvm version 10.0.0.

3 Strengths and Weaknesses

Existing violation witness checkers (excluding Dartagnan [10] designed for concurrent programs) can be roughly divided into two categories.

- CPA-witness2test [6], FShell-witness2test [6], and Nitwit [14] perform one program execution based on the information in the witness. If this execution violates the specification, the witness is confirmed. This approach is very efficient for witnesses fully describing one program execution that violates the property. However, if a witness describes more program executions and only some of them violate the property, these tools can easily miss the violating executions. In particular, if a witness does not specify some return value of a `__VERIFIER_nondet_*` function, FShell-witness2test uses the default value 0, Nitwit picks a random value, and CPA-witness2test fails the witness confirmation.

– CPAchecker [5], UltimateAutomizer [5], and MetaVal [7] create a product of a given witness automaton and the original program and analyze it. As a result, some execution paths of the original program can be analyzed repeatedly for different paths in the witness automaton. To suppress this effect, these checkers usually ignore the possibility to stay in a witness automaton node whenever there is a matching transition leaving the node. Unfortunately, a valid witness can be unconfirmed due to this strategy.

We believe that our approach to checking violation witnesses removes all mentioned disadvantages. Symbolic execution allows us to efficiently examine many program executions corresponding to a given witness automaton, and program executions are not analyzed repeatedly. The approach can easily handle witnesses based on return values from the `__VERIFIER_nondet_*` functions as well as those based on description of branching.

There is only one principal case when a valid witness is not confirmed by Symbiotic-Witch (ignoring the cases when Symbiotic-Witch simply runs out of resources). This case can arise when Symbiotic-Witch uses the information about return values of `__VERIFIER_nondet_*` functions stored in the witness. Symbiotic-Witch uses the information immediately when the symbolic execution calls such a function and there is a matching edge in the witness with a return value that has not been used yet (i.e., the starting node of the edge is in the set of tracked witness nodes and the target node is not). This "eager approach" usually works very well, especially for witnesses containing return values for all calls of `__VERIFIER_nondet_*` functions. However, there can be witnesses where some return values are missing and a particular contained return value should not be used for the first matching call of the `__VERIFIER_nondet_*` function. Such witnesses can be valid, but Symbiotic-Witch can fail to confirm them. As far as we know, such witnesses do not appear in SV-COMP and other witness checkers would probably fail to confirm them as well.

On the negative side, our approach inherits the disadvantages and limitations of symbolic execution and Klee. In particular, it can suffer the *path explosion problem* on witnesses that do not provide return values of `__VERIFIER_nondet_*` functions. Further, Symbiotic-Witch does not support parallel programs as Klee does not support them.

Our current approach is suitable for cases when a witness can be checked based on a finite program execution. That is why our tool supports violation witnesses of safety properties. Table 1 shows the numbers of violation witnesses confirmed in SV-COMP 2022 [2] by individual witness checkers in the categories supported by Symbiotic-Witch.

We believe that symbolic execution can be also used for checking *termination* violation witnesses and for checking correctness witnesses. We plan to extend Symbiotic-Witch in these directions. We also plan to add a witness *refinement mode* [5] already provided by CPAchecker and UltimateAutomizer. In this mode, when a witness is confirmed, Symbiotic-Witch would produce another witness describing a single program execution (by specifying return values for all calls of `__VERIFIER_nondet_*` functions) that exhibits the property violation.

Table 1. The numbers of confirmed witnesses in relevant SV-COMP 2022 categories

	ReachSafety	MemSafety	NoOverflows	SoftwareSystems
number of witnesses	26 797	16 984	2 808	2 102
CPAchecker	14 908	**12 594**	2 334	**621**
CPA-witness2test	8 628	231	887	6
FShell-witness2test	14 168	954	1 436	33
MetaVal	0	116	1 982	0
Nitwit	**15 507**	-	-	0
Symbiotic-Witch	11 176	8 394	**2 609**	179
UltimateAutomizer	8 592	4 197	2 468	26

4 Tool Setup and Configuration

For the use in SV-COMP 2022, we have integrated our witness checker (originally called Witch-Klee) with Symbiotic [9], which takes care of translation of a given C program into llvm using Clang and then slightly modifies the llvm program to improve the efficiency of witness checking.

The archive with Symbiotic-Witch can be downloaded from SV-COMP archives. The witness checking process is invoked by

```
./symbiotic [-prp <prop>] [-32] -witness-check <wit.graphml> <prog.c>
```

where `<wit.graphml>` is a violation witness to be checked and `<prog.c>` is the corresponding program. By default, the tool considers *reachability safety* property and 64-bit architecture. The considered property can be changed by the `-prp` option and `<prop>` instantiated to `memsafety` or `memcleanup` or `no-overflow`. The 32-bit architecture is set by `-32`.

Our witness checker can be also downloaded directly from its repository mentioned below. The version used in SV-COMP 2022 is marked with the tag *SV-COMP22*. It can be executed without Symbiotic via a shell script as

```
./witch.sh <prog.c> <wit.graphml>
```

which calls Clang to translate `<prog.c>` to llvm and then passes the llvm program and the witness `<wit.graphml>` to the witness checker.

5 Software Project and Contributors

Symbiotic-Witch has been developed at Faculty of Informatics, Masaryk University by Paulína Ayaziová under the guidance of Marek Chalupa and 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 our witness checker can be found at:

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

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [2] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of Symbiotic-Witch used in the competition is archived together with other participating tools [3].

References

1. Ayaziová, P.: Klee-based error witness checker. Bachelor's thesis, Masaryk University (2021), https://is.muni.cz/th/rnv19/?lang=en
2. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
3. 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
4. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. (2022), to appear.
5. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Nitto, E.D., Harman, M., Heymans, P. (eds.) Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015. pp. 721–733. ACM (2015), https://doi.org/10.1145/2786805.2786867
6. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses - execution-based validation of verification results. In: Dubois, C., Wolff, B. (eds.) Tests and Proofs - 12th International Conference, TAP@STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10889, pp. 3–23. Springer (2018), https://doi.org/10.1007/978-3-319-92994-1_1
7. Beyer, D., Spiessl, M.: Metaval: Witness validation via verification. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 165–177. Springer (2020), https://doi.org/10.1007/978-3-030-53291-8_10
8. 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
9. Chalupa, M., Řechtáčková, A., Mihalkovič, V., Zaoral, L., Strejček, J.: Symbiotic 9: Parallelism and invariants (competition contribution). In: Proc. TACAS (2). Springer (2022)
10. Haas, T., Meyer, R., de León, H.P.: Dartagnan: SMT-based violation witness validation (competition contribution). In: Proc. TACAS (2). Springer (2022)
11. King, J.C.: Symbolic execution and program testing. Communications of ACM **19**(7), 385–394 (1976), https://doi.org/10.1145/360248.360252
12. 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
13. 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

14. Švejda, J., Berger, P., Katoen, J.: Interpretation-based violation witness validation for C: NITWIT. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12078, pp. 40–57. Springer (2020), https://doi.org/10.1007/978-3-030-45190-5_3