

Witch 3: Validation of Violation Witnesses in the Witness Format 2.0*

(Competition Contribution)

Paulína Ayaziová(✉)** and Jan Strejček

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

Abstract. Witch 3 is a new validator of violation witnesses in the witness format 2.0. Note that our previous tool, Symbiotic-Witch 2, can validate only violation witnesses in the old GraphML format. Witch 3 validates witnesses of reachability of an error function, overflows, and invalid dereferences and deallocations. Similarly to Symbiotic-Witch 2, the tool is based on symbolic execution and uses parts of the Symbiotic framework. Support of the witness format 2.0 in Witch 3 includes features not supported by Symbiotic-Witch 2, such as constraints on the program variables and function return values, specifying statements by column, and providing the concrete statement in which the violation occurs. These additional features can further restrict the explored state space, and, more importantly, allow for much more precise validation.

1 Witness Validation Approach

Witch 3 is a new validator of violation witnesses based on symbolic execution. It extends the line of validators Symbiotic-Witch [1] and Symbiotic-Witch 2 [2], which are used to validate violation witnesses in the GraphML witness format [6] (now called 1.0). The main difference of Witch 3 is that it processes witnesses in the witness format 2.0[1] [3] (also known as "the YAML format"). Since this format is based on witness segments and waypoints as opposed to witness automata from the GraphML format, there are large differences in the validation process compared to Symbiotic-Witch 2.

Since the tool performs symbolic execution on the llvm IR [9] of the input program and some information may be lost during the compilation, we first preprocess both the witness and the input program. The preprocessing entails wrapping the branching conditions in the program with a special function so that the condition is not decomposed or flipped during compilation. This ensures that the conditions in the branching statements and the corresponding branches are correctly mapped to those described in the witness. Another crucial step is adjusting the witness so that the identifiers of the waypoints will be preserved

* This work has been supported by the Czech Science Foundation grant GA23-06506S.
** Jury member of SV-COMP 2024
[1] Description is available at https://gitlab.com/sosy-lab/benchmarking/sv-witnesses.

© The Author(s) 2024
B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 341–346, 2024.
https://doi.org/10.1007/978-3-031-57256-2_18

in the debug information in the LLVM program. In this phase we also inject the constraints from the assumption waypoints into the input program as calls to a special function `__VALIDATOR_assume` which will be handled later. After this preprocessing, the tool compiles the program into LLVM IR and runs the internal validator WITCH-KLEE on the LLVM program and the preprocessed witness.

The validator begins symbolic execution in the entry point of the program, associating this state with the first segment of the witness. Throughout the process, each state of symbolic execution is associated with one witness segment.

Whenever the tool executes an instruction that could be associated with a waypoint of type `function_enter`, `function_return`, or `branching` (i.e. a function call, function return, or a branching instruction), it is checked whether this instruction matches a waypoint of the associated segment and the corresponding constraint is enforced on the state. More precisely, if the instruction matches the type and location of an avoid waypoint in the segment, the negation of the constraint in the witness is added to the path condition of the state to guarantee that the waypoint is avoided. If the path condition is not satisfiable, the current state of symbolic execution is terminated. Note that this is always the case for waypoints of type `function_enter`, as their fixed constraint *true* is negated into *false*. If the instruction matches the follow waypoint of a segment, we add the given constraint to the path condition and the witness traversal moves to the next segment.

The `assumption` waypoints are handled slightly differently. Since the constraints are already injected in the program, what remains is to enforce them at the right time. Hence, whenever a `__VALIDATOR_assume` call is executed, the tool checks whether the current state of symbolic execution is associated with the corresponding segment. If it is not, the call is ignored and symbolic execution continues normally. Otherwise, for a follow waypoint, the tool adds the constraint to the path condition of the state and moves to the next segment. For an avoid waypoint, we enforce the negation of the constraint in a similar manner. If the resulting path condition is not satisfiable, the state is terminated.

If the symbolic executor detects a property violation, the tool investigates whether the violating instruction matches the `target` waypoint, which is the last waypoint of the violation witness. If the segment associated with the violating state is not the last, the tool terminates the current state but continues exploring other states of symbolic execution. This is also the case if the associated segment is the last but the `target` waypoint of the segment does not match the instruction violating the property. Otherwise, i.e., if the witness traversal reached the target waypoint, WITCH 3 confirms the witness by reporting `false`.

If the exploration ends without confirming the witness, there are two possible results. Normally, WITCH 3 outputs `true` to refute the witness. However, the symbolic executor used by WITCH 3 may replace a symbolic value by a concrete one due to an unsupported feature (for example, it does not support symbolic floats). This substitution can cause that not all possible execution paths are explored and thus a valid witness can be refuted. Hence, in such instances, witness refutation is suppressed and WITCH 3 reports `unknown`.

2 Strengths and Weaknesses

The main strong point of Witch 3 is the support of all features of the format 2.0. This includes enforcing constraints on the values of program variables. These constraints can be included also in the GraphML witnesses, but they are ignored by both Symbiotic-Witch and Symbiotic-Witch 2 with the exception of the equality constraints on the return values of `__VERIFIER_nondet_*` functions. These tools also ignore the attribute `offset` (replaced by `column` in the witness format 2.0) specifying the exact location of an instruction on a given program line. Such shortcomings of our older validators can lead to incorrectly validated witnesses and more `unknown` results. In contrast, full support of the new format allows Witch 3 to produce much more reliable results. Moreover, even in the cases where our older validators produce a correct result without using the contraints provided in the witness, Witch 3 can use the constraints to reduce the explored state-space and thus speed up the validation.

On the negative side, the witness format 2.0 currently supports only witnesses of reachability of an error function, overflows, and invalid dereferences and deallocations. Hence, Witch 3 can only be used in these categories. Once the format is extended for more properties, we plan to implement their support.

Another shortcoming is that the tool currently requires the exact location of a waypoint, including the optional column number. This does not cause any incorrect results since the validation fails in the case of missing information. Moreover, as of SV-COMP 2024, all tools which produced violation witnesses in the format 2.0 included this information. Despite this, we consider it a weakness and plan to fix it in future versions of the tool.

Witch 3 also inherits weaknesses from the technology that it uses. The fact that the symbolic executor works with programs in llvm requires more preprocessing on the program and the witness to ensure that no crucial information is lost during the translation. For this reason, there are cases in which the validation process may be slower. Additionally, the program may contain some inner nondeterminism, such as an unspecified order of evaluation, which is resolved during the compilation. If this order is different to that prescribed by the witness, the witness may be incorrectly refuted. However, we have not yet found any such incorrect result in practice. Most incorrect results stem from technical errors such as missing models of library functions — these functions are then treated as nondeterministic and pure, which may not be the case.

3 Software Architecture

Witch 3 can be divided into two components: Symbiotic [8], which is used as a wrapper for the second component, and Witch-Klee, a witness validator for llvm programs.

For the purposes of this tool, we extended Symbiotic 9 with scripts for preprocessing the witness and the program as previously described. It also compiles the program into llvm, links necessary function models, and parses the output of the internal validator, Witch-Klee.

Witch-Klee takes the program in the llvm IR and the preprocessed witness and performs the validation. The tool is based on the symbolic executor JetKlee, a fork of Klee [7] developed for the purposes of Symbiotic. Witch-Klee uses the yaml-cpp[2] library to parse the witness in the YAML format and Z3 [10] as the SMT solver in symbolic execution.

Both components of Witch 3 use llvm 10.0.1.

4 Tool Setup and Configuration

The archive containing Witch 3 as it participated in SV-COMP 2024 is available on Zenodo [4]. The validation is invoked by the command

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

where `<prop>` is the considered property, the switches `--32` and `--64` specify the considered architecture, `<witness>` is a violation witness in the YAML format, and `<prg>` is the input C program. The property can be provided either as a `.prp` file or one of the shortcuts `no-overflow` and `valid-memsafety`. The default setting is the property of unreachability of the function `reach_error` and the 64-bit architecture.

The version of Symbiotic used by Witch 3, as well as the internal validator Witch-Klee, are available on GitHub (see below) under the tag `svcomp24`. To build Witch 3 from its sources, build each of the components separately. To run the validator, add the location of the Witch-Klee executable to `$PATH` and use the command as presented above.

5 Software Project and Contributors

Both components of Witch 3 are available on GitHub. The source code of the validator Witch-Klee is available at

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

and the source code of the version of Symbiotic used by Witch 3 can be found at

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

The tool has been developed at the Faculty of Informatics of Masaryk University by Paulína Ayaziová under the supervision and with advice of Jan Strejček. It is available under the MIT license and all internally used tools and libraries (llvm, JetKlee, Z3, yaml-cpp, Symbiotic) are available under open-source licenses that comply with SV-COMP's policy for the reproduction of results.

[2] https://github.com/jbeder/yaml-cpp

Data Availability Statement. All data of SV-COMP 2024 are archived as described in the competition report [5] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of Witch 3 used in the competition is archived on Zenodo [4].

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: More efficient algorithm and witness refutation (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13994, pp. 523–528. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_30
3. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. Submitted to SPIN 2024.
4. Ayaziová, P., Strejček, J.: Witch 3. Zenodo (2023). https://doi.org/10.5281/zenodo.10064512
5. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Finkbeiner, B., Kovács, L. (eds.) TACAS 2024. LNCS, vol. 14572, pp. xx–yy. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-57256-2_15
6. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579, https://doi.org/10.1145/3477579
7. 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
8. 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.) 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
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

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.