Symbiotic 4: Beyond Reachability
(Competition Contribution)

Marek Chalupa(✉), Martina Vitovská, Martin Jonáš, Jiri Slaby, and Jan Strejček

Faculty of Informatics, Masaryk University, Brno, Czech Republic
xchalup4@fi.muni.cz

Abstract. The fourth version of Symbiotic brings a brand new instrumentation part, which can now instrument the analyzed program with code pieces checking various specification properties. As a consequence, Symbiotic 4 participates for the first time also in categories focused on memory safety. Further, we have ported both Symbiotic and Klee to llvm 3.8 and added new features to the slicer which is now modular and easily extensible.

1 Verification Approach and Software Architecture

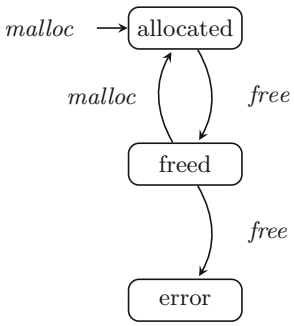
Symbiotic implements the approach of [6] combining instrumentation, slicing, and symbolic execution [4] to detect errors in C programs. While all the previous releases [2,5,7] focus on checking reachability of an error location, Symbiotic 4 can check any property definable by a finite state machine. For example, the finite state machine of Fig. 1 describes the double free error. Intuitively, for every allocated block of memory we create a copy of the state machine that tracks its current status. An error state is reached if the block is deallocated twice. Hence, the instrumentation reduces property checking to unreachability checking as the program violates the property iff the error state is reachable.

Creation and tracking of the state machine is performed by code instrumented to the original program. In fact, the brand new instrumentation implemented in Symbiotic works more generally. It gets a JSON file with instrumentation rules. Every rule specifies a function call to be inserted before (or after) each occurrence of a given sequence of instructions. Bodies of called functions are then defined in a separate file written in C. Each instrumentation rule can be refined using an output of a specified static analysis. For example, a code checking NULL dereference does not have to be instrumented to locations where a suitable static analysis guarantees that the corresponding pointer cannot be NULL.

For SV-COMP 2017, we have prepared instrumentation rules for checking *memory safety* properties. For *overflow* property, we let clang sanitizer to

The research was supported by The Czech Science Foundation, grant GA15-17564S.

A. Legay and T. Margaria (Eds.): TACAS 2017, Part II, LNCS 10206, pp. 385–389, 2017.
DOI: 10.1007/978-3-662-54580-5_28




```
int *p = malloc(sizeof(int));
fsm_change_state(p, FSM_MALLOC);
⋮
fsm_change_state(p, FSM_FREE);
free(p);
⋮
if (__VERIFIER_nondet_int()) {
   fsm_change_state(p2, FSM_FREE);
   free(p2); // double free if p2==p
}
```

Fig. 1. State machine describing double free and a code example with instrumented function calls (red). (Color figure online)

instrument the program. We do not support checking *termination* property as it cannot be simply translated to reachability analysis.

The workflow of Symbiotic 4 is illustrated by Fig. 2. As the first step, we check that the verified property is not *termination*. Then we translate the analyzed C program to the llvm bitcode by clang. Next, we check that the bitcode contains no calls to `pthread_create` as neither our slicer, nor Klee can process concurrent programs. If the check is successful, we proceed to the instrumentation of the bitcode. The instrumentation step has two phases. In the first phase, we insert instructions that tell the symbolic executor to treat all memory as symbolic, which allows us to correctly handle uninitialized variables. In the second phase, we perform a static analysis of the bitcode and instrument it as described above. We currently use a points-to analysis when instrumenting *memory safety* properties to insert property-checking functions only to the locations where the analysis itself does not guarantee that the property holds. The inserted functions call `__VERIFIER_error` whenever the property is violated. Definitions of the inserted property-checking functions as well as definitions of `__VERIFIER_*` functions are then linked to the bitcode. Parts of the produced code that have no effect on reaching `__VERIFIER_error` call sites are consequently removed by slicing. Moreover, code optimizations provided by llvm are used before and after slicing. Before the bitcode is symbolically executed by Klee [1], we check that it does not contain instructions related to the *floating point arithmetic* not supported by Klee, e.g. `__isnan` or `__inf`. We use our fork of Klee that produces an error witness when a property violation is detected. If Klee reports that `__VERIFIER_error` is unreachable, we return `true` and a trivial correctness witness unless Klee warns about not exploring the whole state space. This can happen for example due to limitted support of floating point instructions. In such cases, we return `unknown`.

The slicer has undergone significant changes. Points-to analyses and reaching definitions analysis (needed to build dependency graphs for slicing [3]) were redesigned into a more general modular framework: Symbiotic now supports

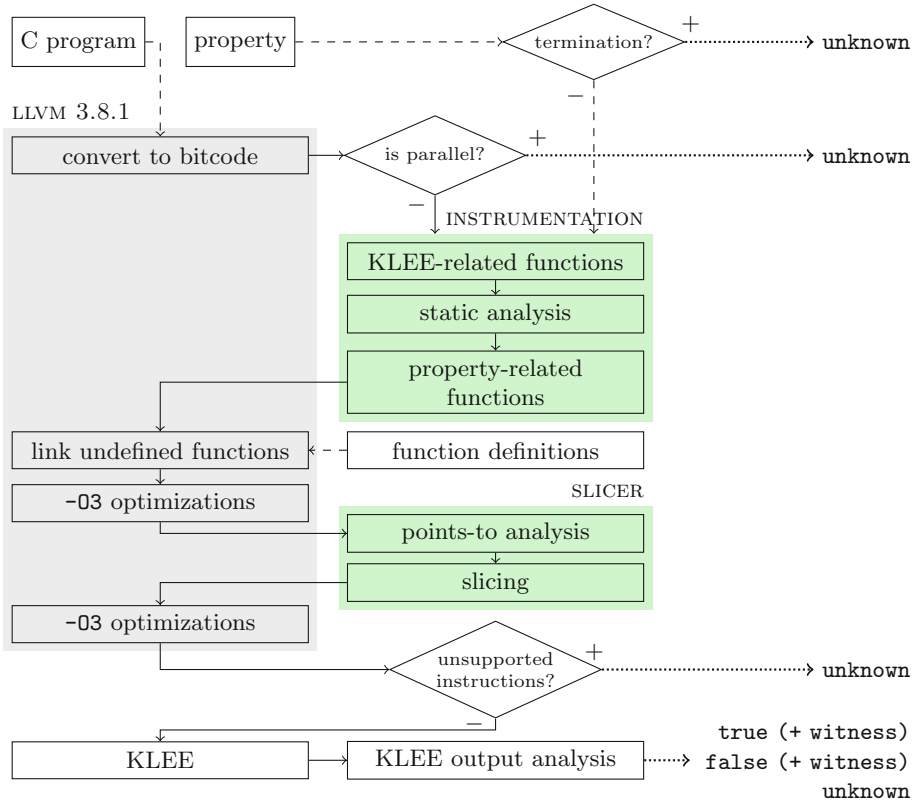


Fig. 2. Workflow of Symbiotic 4. Dashed lines represent verification inputs, solid lines llvm bitcode and control flow, and dotted lines represent outputs.

more types of analyses that share a common interface and are therefore interchangeable. In particular, the current version of Symbiotic supports both flow-sensitive and flow-insensitive points-to analyses and for both of these analyses, field-sensitive and field-insensitive variants are available. Further, points-to analyses can now precisely handle a larger subset of llvm including `memset` and `memcpy` llvm's intrinsic calls. We have also implemented additional optimizations based on the information about strongly connected components of the program's control flow graph to speed up the analyses. Note that the redesigned analyses are not firmly integrated into the slicer and can therefore be reused by external tools.

The last significant change in Symbiotic 4 is that all components have been ported to llvm 3.8, including the symbolic executor Klee. Finally, we got rid of separate Perl and bash scripts in favor of a concise modular implementation in Python.

2 Strengths and Weaknesses

The main strength of the approach is its universality and modularity. Thanks to the instrumentation, Symbiotic now supports almost all checked properties specified by SV-COMP. Authors of other llvm-based verification tools can also benefit from the implemented instrumentation and slicer: the instrumentation can be used to add the ability to verify additional properties such as memory safety to tools that only support reachability and the slicer can be used to remove irrelevant parts of the verified program.

The main disadvantage of the current configuration is the high computational cost of symbolic execution for branching-intensive programs. However, thanks to the modular architecture, a suitable software verifier can be in principle used instead of Klee to alleviate this problem.

3 Tool Setup and Configuration

- *Download:* https://github.com/staticafi/symbiotic/releases/tag/4.0.0
- *Installation:* Unpack the archive. The only requirement is `python 2.7`.
- *Participation Statement:* Symbiotic 4 participates in all categories.
- *Execution:* Run `./symbiotic OPTS <source>`, where available `OPTS` include:
  - `--64`, which sets the environment for 64-bit benchmarks,
  - `--prp=file`, which sets the property specification file to use,
  - `--witness=file`, which sets the output file for the witness,
  - `--help`, which shows the full list of possible options.

4 Software Project and Contributors

Symbiotic 4 has been developed by M. Chalupa, M. Vitovská, and J. Slaby with support of M. Jonáš and under supervision of J. Strejček. The tool and its components are available under GNU GPLv2 and MIT Licenses. The project is hosted by the Faculty of Informatics, Masaryk University. llvm, Klee, stp, and MiniSat are also available under open-source licenses. The project web page is: https://github.com/staticafi/symbiotic

References

1. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI, pp. 209–224. USENIX Association (2008)
2. Chalupa, M., Jonáš, M., Slaby, J., Strejček, J., Vitovská, M.: Symbiotic 3: new slicer and error-witness generation. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 946–949. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_67
3. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990)

4. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
5. Slaby, J., Strejček, J.: Symbiotic 2: more precise slicing. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 415–417. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54862-8_34
6. Slabý, J., Strejček, J., Trtík, M.: Checking properties described by state machines: on synergy of instrumentation, slicing, and symbolic execution. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 207–221. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32469-7_14
7. Slaby, J., Strejček, J., Trtík, M.: Symbiotic: synergy of instrumentation, slicing, and symbolic execution. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 630–632. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36742-7_50