Symbiotic 2: More Precise Slicing*
(Competition Contribution)

Jiri Slaby and Jan Strejček

Faculty of Informatics, Masaryk University
Botanická 68a, 60200 Brno, Czech Republic
{slaby,strejcek}@fi.muni.cz

Abstract. SYMBIOTIC 2 keeps the concept and the structure of the original bug-finding tool SYMBIOTIC, but it uses a more precise slicing based on a field-sensitive pointer analysis instead of field-insensitive analysis of the original tool. The paper discusses this improvement and its consequences. We also briefly recall basic principles of the tool, its strong and weak points, installation, and running instructions. Finally, we comment the results achieved by SYMBIOTIC 2 in the competition.

1 Verification Approach and Software Architecture

Both SYMBIOTIC [6] and SYMBIOTIC 2 implement our verification approach proposed earlier [4]. The approach combines three standard techniques, namely *code instrumentation*, *program slicing* [7], and *symbolic execution* [3]. While the approach was originally designed for the detection of bugs described by state-machines, SYMBIOTIC 2 still supports only one kind of bugs: reachability of an `ERROR` label. Hence, we briefly recall the approach restricted just to this simple kind of errors. We explain the structure of the tool simultaneously.

1. Code instrumentation inserts `assert(0)` to each `ERROR` label. It is performed by a `bash` script calling `sed`. The instrumented code is translated into the LLVM bitcode by the CLANG compiler.
2. Program slicing removes instructions of the instrumented code that do not affect reachability of the inserted `assert(0)` statements. This code size reduction is crucial for the overall efficiency of SYMBIOTIC 2. The slicer is implemented in C++ as a plug-in for the LLVM optimizer `opt`.
3. Symbolic execution either reaches `assert(0)`, or correctly finishes the execution without reaching `assert(0)`, or it runs out of time or memory etc. These possibilities correspond to answers `FALSE`, `TRUE`, `UNKNOWN`, respectively. We use the symbolic executor KLEE [2] whose outputs are translated to `TRUE/FALSE/UNKNOWN` by a simple `bash` script.

The whole pipeline is executed stepwise by another `bash` script.

All improvements of SYMBIOTIC 2 over the tool SYMBIOTIC competing in SV-COMP 2013 are in the slicer. We have fixed some bugs in the original slicer

* The authors are supported by the Czech Science Foundation grant P202-10-1469.

(invalid treatment of several instructions and functions with variable number of arguments). While the original fixpoint algorithm for slicing [7] is relatively simple, it gets more complicated for programs with pointers as one instruction can influence the following one without any syntactic overlap. We need to use a pointer analysis (also called points-to analysis) to know which pointers can point to the same target. Both Symbiotic and Symbiotic 2 use Andersen's pointer analysis [1], Symbiotic 2 replaces the original field-insensitive analysis by a field-sensitive one. This means that every field of a `struct` is now handled as a different pointer target. Similarly, we handle the first 64 elements of each array as distinct targets. The field-sensitive analysis is computationally more demanding. Thus we have also added some type filters that speed up the pointer analysis and make its results more accurate. All these improvements of the slicer significantly reduce the number of incorrect answers produced by Symbiotic 2. For example, Symbiotic 2 produces only correct answers for the category *ProductLines* while Symbiotic produces 131 incorrect answers for the same category.

2 Strengths and Weaknesses

Our tool is applicable to all competition benchmarks satisfying two restrictions: the studied property is `ERROR` label reachability and the benchmark code is a sequential C program. Hence, the results of Symbiotic 2 in the competition categories *MemorySafety* and *Concurrency* should be ignored (we missed the opt-out deadline). The first restriction can be removed by the implementation of a more sophisticated code instrumentation. The second restriction comes directly from the approach as symbolic execution and program slicing are primarily designed for sequential programs.

We first discuss strong and weak aspects of the approach and then we talk about additional strong and weak aspects of the tool. Our approach is based on symbolic execution which should produce only correct answers. On the other hand, symbolic execution suffers from the path explosion problem and relies on expensive (and often even undecidable) SMT solving. Hence, application of symbolic execution leads to many `UNKNOWN` answers, which is also the main weakness of the approach. To reduce this weakness, we combine symbolic execution with slicing which is the only theoretical source of incorrect answers (namely false positives) of our approach. Indeed, slicing can in some cases remove an infinite loop and a potential unreachable `ERROR` label located below the cycle thus become reachable. However, this situation is very rare in practice (e.g. it does not appear in the competition benchmarks) and we do not see it as a problem. An orthogonal method to reduce the high cost of symbolic execution is to use some of its variants suppressing the path explosion problem. For example, we plan to apply *compact symbolic execution* [5] instead of the classic one.

The strong aspect of the tool is its simple architecture: it is a sequence of scripts and standalone tools that are easy to replace (for example, if there is a better symbolic executor for Llvm bitcode, we can deploy it in few minutes). The main weakness of Symbiotic 2 lies in the incorrect results which are sometimes reported. Even if the number of incorrect results is substantially lower than in the case of Symbiotic, it is still relatively high. All the incorrect results are due to imperfection of our implementation.

3 Tool Setup and Configuration

Before using Symbiotic 2, ensure that the target system contains 32-bit libraries (for 32-bit benchmarks) and Llvm with Clang. Llvm and Clang have to be in version 3.2 exactly. Then, Symbiotic 2 can be downloaded from `http://sf.net/projects/symbiotic/`. Due to a bug in Klee causing absolute paths to be built in, Klee requires to be run from a pre-defined path. Hence we are obliged to change the current directory to `/opt/` and *untar* the downloaded Symbiotic 2 archive there. Running the tool is then straightforward. When the current directory is `/opt/symbiotic/`, the tool can be invoked for each `<benchmark.c>` from the set by `./runme <benchmark.c>`. For benchmarks intended for 64-bit, set `MFLAG=-m64` environment variable. The answers provided by Symbiotic 2 are as required by the competition rules: `TRUE/FALSE/UNKWNOWN`. If the result for `<benchmark.c>` is `FALSE`, discovered error paths can be found in `<benchmark.c>-klee-out/`.

4 Software Project and Contributors

Symbiotic 2 was contributed mostly by the authors of this paper and Marek Trtík. Jiri Slaby is a contact person. The tool is licensed under the GNU GPLv2 License unless specified otherwise for its parts.

References

1. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.
2. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI*, pages 209–224. USENIX Association, 2008.
3. J. C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.
4. J. Slabý, J. Strejček, and M. Trtík. Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In *Proceedings of FMICS*, volume 7437 of *LNCS*, pages 207–221. Springer, 2012.
5. J. Slaby, J. Strejček, and M. Trtík. Compact symbolic execution. In *Proceedings of ATVA*, volume 8172 of *LNCS*, pages 193–207. Springer, 2013.
6. J. Slaby, J. Strejček, and M. Trtík. Symbiotic: Synergy of instrumentation, slicing, and symbolic execution - (competition contribution). In *Proceedings of TACAS*, volume 7795 of *LNCS*, pages 630–632. Springer, 2013.
7. M. Weiser. Program slicing. In *Proceedings of ICSE*, pages 439–449. IEEE, 1981.