Symbiotic 5: Boosted Instrumentation*
(Competition Contribution)

Marek Chalupa**, Martina Vitovská, and Jan Strejček

Masaryk University, Brno, Czech Republic

Abstract. The fifth version of Symbiotic significantly improves instrumentation capabilities that the tool uses to participate in the category *MemSafety*. It leverages an extended pointer analysis re-designed for instrumenting programs with memory safety errors, and staged instrumentation reducing the number of inserted function calls that track or check the memory state. Apart from various bugfixes, we have ported Symbiotic (including the external symbolic executor Klee) to llvm 3.9 and improved the generation of violation witnesses by providing values of some variables.

1 Verification Approach

The basic approach of Symbiotic remains unchanged [7]: it uses instrumentation to reduce checking of specific properties (e.g. *no-overflow* or *memory safety*) to checking reachability of error locations. Then we apply slicing which removes the code that has no influence on reachability of these locations. Finally, we symbolically execute the sliced code using Klee [1] to refute or confirm that an error location is reachable.

For many years, our attention has been focused mainly on slicing [8,6,2]. Only in 2016, we implemented a configurable instrumentation that enabled Symbiotic to check memory safety or, in general, any safety property. Consequently, Symbiotic 4 [4] participated for the first time in the category *MemSafety* where it won the bronze medal.

The instrumentation used in Symbiotic 4 to check memory safety inserts calls to functions that *track* every block of allocated memory and calls to functions that *check* validity of dereferences using the tracked information. A check is not inserted if a static pointer analysis guarantees that the dereferenced pointer points to a memory block that was allocated before. Later we have recognized a flaw of this optimization: a standard pointer analysis ignores memory deallocations and, hence, it can tell that a pointer can point to memory blocks allocated by specific program lines, but it does not tell whether these memory blocks are *still* allocated. As a result, Symbiotic 4 sometimes does not insert a check even if the dereference may be invalid and thus it may miss some bugs.

* The research is supported by the Czech Science Foundation grant GBP202/12/G061.

** Jury member and corresponding author: xchalup4@fi.muni.cz.

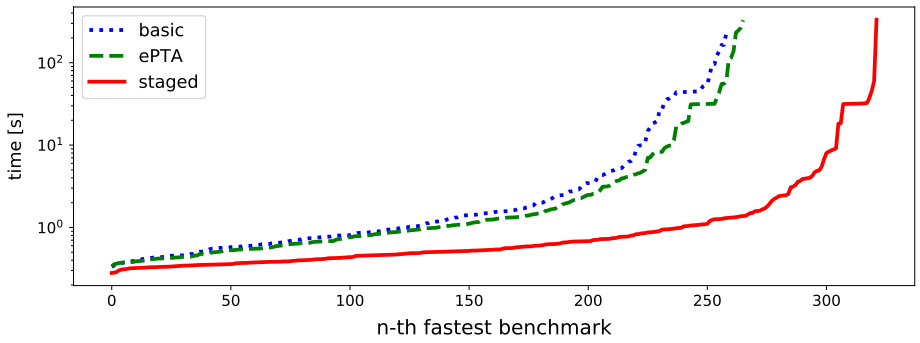

Fig. 1. Quantile plot of running times of the three considered configurations of Symbiotic 5. On the x-axis are the benchmarks sorted according to the corresponding running times and on the logarithmic y-axis are the times.

In Symbiotic 5, we have fixed and significantly boosted the instrumentation part. First, we have extended the above mentioned pointer analysis such that it takes into account deallocations as well. Second, the instrumentation now works in two stages. The first stage inserts the checks where extended pointer analysis cannot guarantee the dereference safety. Moreover, compared to Symbiotic 4, we use simpler checks if possible. For example, if a pointer analysis says that a given pointer points into a known fixed-size memory block, we just insert a check that the pointer's offset is within the size of the block (without searching the tracked information about the block). The second stage inserts calls to memory tracking functions only to allocations of the memory blocks that can be accessed by some dereference instrumented in the first stage. Hence, we track only the information that may be possibly used in the checks.

To evaluate the boosted instrumentation, we run the following three configurations of Symbiotic on 393 benchmarks of the SV-COMP 2017 meta category *MemSafety* and of the category *MemSafety-TerminCrafted*:

- **basic** uses instrumentation without any pointer analysis,
- **ePTA** uses extended pointer analysis (i.e. it is a fixed version of the instrumentation in Symbiotic 4),
- **staged** uses extended pointer analysis and staged instrumentation.

Figure 1 clearly shows that the performance improvement brought by the extended pointer analysis itself is negligible compared to the performance improvement delivered by the extended pointer analysis in combination with staged instrumentation. For a precise description of the boosted instrumentation, experimental setup and results, we refer to [3].

Symbiotic 5 also changed the approach to error witness generation. Symbiotic 4 describes an errorneous run by a sequence of passed program locations. The sequence is often very long and it turned out to be too restrictive for witness

checkers. SYMBIOTIC 5 provides only the starting and target locations of the run and return values of some `__VERIFIER_nondet*` calls. More precisely, we provide return values of calls in `main` and such that they are called just once in the run. The witnesses are now more often confirmed by witness checkers.

2 Software Architecture

All components of SYMBIOTIC are built on top of LLVM 3.9 [9]. We use the CLANG compiler to compile the analyzed sources into LLVM bitcode. Symbiotic consists of scripts written in Python that distribute work to three basic modules, all written in C++:

Instrumentation module This module inserts function calls to instructions according to a given configuration in JSON. The instrumented functions are implemented in C and compiled to LLVM automatically by SYMBIOTIC before the instrumentation process. We use this configurable instrumentation for instrumenting the *memory safety* property only. For instrumenting the *no-overflow* property, we use CLANG's sanitizer as it works sufficiently well in this case.

Slicing module This module implements an interprocedural version of the slicing algorithm based on dependence graphs [5] altogether with analyses that are needed to compute dependencies between instructions, i.e. pointer analyses (including the extended pointer analysis as described in Section 1 that is used by the instrumentation) and analyses of reaching definitions.

Verification backend For deciding reachability of error locations, we currently use our clone of the open-source symbolic executor KLEE [1], that was ported to LLVM 3.9 and modified to support error witness generation.

Before and after slicing, we optimize the code using available LLVM's optimizations. The rest of bitcode transformations that we use and whose nature is mostly technical (e.g. replacement of calls inserted by CLANG's sanitizer to `__VERIFIER_error` calls) are implemented as LLVM passes. All the components that transform bitcode take a bitcode as an input and give a valid bitcode as an output. This makes SYMBIOTIC highly modular: any part (module) can be easily replaced or used as a stand-alone tool.

3 Strengths and Weaknesses

The main strength of the approach is its universality and modularity. The instrumentation can reduce any safety property to reachability checks and therefore no special monitors need to be incorporated into the verification backend. Indeed, any tool that can decide reachability of error locations can be plugged-in.

The main disadvantage of the current configuration is that symbolic execution does not satisfactory handle programs with unbounded loops. Moreover, KLEE cannot generate invariants for loops.

4 Tool Setup and Configuration

- *Download:* `https://github.com/staticafi/symbiotic/releases/download/5.0.1/symbiotic-5.0.1.zip`
- *Installation:* Unpack the archive.
- *Participation Statement:* SYMBIOTIC 5 participates in all categories.
- *Execution:* Run `bin/symbiotic OPTS <source>`, where available `OPTS` include:
  - `--prp=file`, which sets the property specification file to use,
  - `--witness=file`, which sets the output file for the witness,
  - `--32`, which sets the 32-bit environment,
  - `--help`, which shows the full list of possible options.

5 Software Project and Contributors

SYMBIOTIC 5 has been developed by M. Chalupa and M. Vitovská 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 and KLEE are also available under open-source licenses. The project web page is: `https://github.com/staticafi/symbiotic`

References

1. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX Association, 2008.
2. M. Chalupa, M. Jonáš, J. Slaby, J. Strejček, and M. Vitovská. Symbiotic 3: New slicer and error-witness generation - (competition contribution). In *TACAS*, volume 8413 of *LNCS*, pages 946–949. Springer, 2016.
3. M. Chalupa, J. Strejček, and M. Vitovská. Joint forces for memory safety checking. Submitted to SPIN 2018.
4. M. Chalupa, M. Vitovská, M. Jonáš, J. Slaby, and J. Strejček. Symbiotic 4: Beyond reachability - (competition contribution). In *TACAS*, volume 10206 of *LNCS*, pages 385–389, 2017.
5. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. In *International Symposium on Programming*, volume 167 of *LNCS*, pages 125–132. Springer, 1984.
6. J. Slaby and J. Strejček. Symbiotic 2: More precise slicing - (competition contribution). In *TACAS*, volume 8413 of *LNCS*, pages 415–417. Springer, 2014.
7. J. Slaby, J. Strejček, and M. Trtík. Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In *FMICS*, volume 7437 of *LNCS*, pages 207–221. Springer, 2012.
8. J. Slaby, J. Strejček, and M. Trtík. Symbiotic: Synergy of instrumentation, slicing, and symbolic execution - (competition contribution). In *TACAS*, volume 7795 of *LNCS*, pages 630–632. Springer, 2013.
9. LLVM. `http://llvm.org/`.