Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Symbiotic 6: Generating Test-Cases by Slicing and Symbolic Execution

Marek Chalupa, Martina Vitovská, Tomáš Jašek, Michael Šimáček, Jan Strejček

Masaryk University, Brno, Czech Republic
Corresponding author: Marek Chalupa (chalupa@fi.muni.cz)
Jury member: Martina Vitovská (vitoma@mail.muni.cz)

The date of receipt and acceptance will be inserted by the editor

Abstract. Symbiotic is a bug-finding and verification tool that integrates light-weight static analyses and instrumentation with program slicing and symbolic execution. The techniques are suitably combined according to a given goal. The paper describes a particular configuration competing in Test-Comp 2019. We also provide a brief analysis of Symbiotic's results achieved in the competition. As our tool uses a fork of the open-source symbolic executor Klee, we focus on comparison with mainstream Klee that also participated in the competition this year.

1 Test-Generation Approach

Symbiotic [9,3] is an open-source bug-finding and verification tool combining configurable code instrumentation (supported by static analyses) [10,2], program slicing, and symbolic execution. The instrumentation modifies a given program by adding error locations that are reachable if and only if the original program contains a selected kind of errors like signed-integer overflow or invalid pointer dereference. Program slicing reduces the instrumented program by removing irrelevant instructions. Finally, symbolic execution decides whether some error location is reachable. As Test-Comp specifies errors as reachable calls of `__VERIFIER_error` function, instrumentation is not applied in the competition configuration. We briefly describe the remaining two techniques.

1.1 Program Slicing

Static (backward) program slicing [11] is a technique that removes program instructions that have no influence on reachability or the effect of selected parts of the program. These parts are called *slicing criteria* and the reduced program is called a *slice*. In the case of Symbiotic's particiaption in Test-Comp, the slicing criteria are all calls to the `__VERIFIER_error` function.

Program slicing is commonly done using *dependence graphs* [5,7]. A dependence graph is a directed graph that contains instructions of the program as nodes, and edges capturing dependencies between instructions. A slice of the program is obtained by a backward search from the nodes that represent the slicing critera, i.e., the slice is formed by all nodes that are backward-reachable from the slicing criteria nodes in the dependence graph.

1.2 Symbolic Execution

Symbolic execution [8] is a program analysis technique that executes the program using symbols instead of concrete values. When a branching instruction is reached, symbolic execution asks an SMT solver which branch it should take. In the case that both branches can be taken, symbolic execution forks and follows both branches. For each program path, symbolic execution builds a *path condition*, which is a collection of all constraints on the symbols that must be satisfied to follow the path. When symbolic execution reaches the end of a path, it uses the corresponding path condition to generate a test-case that makes the program follow this path.

1.3 Workflow

The general workflow of Symbiotic 6 for Test-Comp 2019 is depicted in Figure 1. In the competition configuration, Symbiotic operates in two modes according to the provided property.

For the property `coverage-branches` (where the goal is to generate tests to maximize the coverage), we directly run symbolic execution to generate tests.

For the property `coverage-error-call` (where the goal is to find a test-case that hits an error), we slice

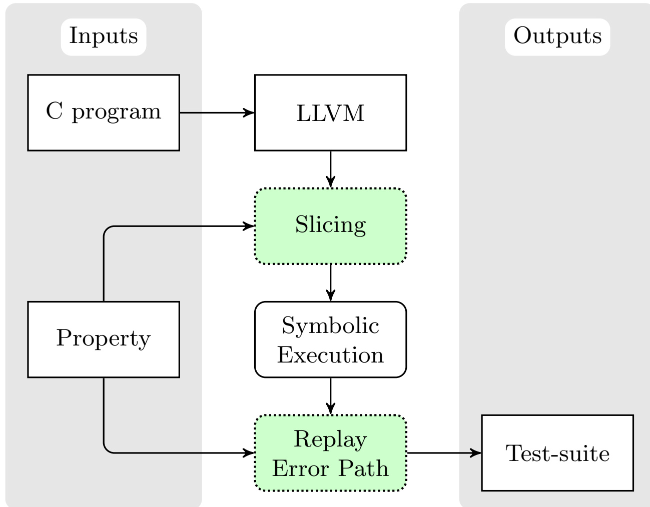

Fig. 1. The workflow of Symbiotic for Test-Comp. Input is compiled into llvm and symbolically executed. If the property is `coverage-error-call`, slicing and error path replaying are enabled (highlighted with dotted border). Finally, test-suite is generated from the output of the symbolic execution.

the analyzed program with respect to calls of the function `__VERIFIER_error`. Consequently, if a feasible path on which this function is called is found by symbolic execution, we attempt to replay this path in the unsliced program. This is important because some calls to `__VERIFIER_nondet_*` functions may have been sliced away and their return values would be missing in the generated tests.

2 Software Architecture

All parts of Symbiotic use llvm framework [12] (in Symbiotic 6 we use llvm version 4.0.1). Therefore the first step is to compile the source code of the analyzed program into llvm bitcode. This is done by the compiler Clang.

To carry out symbolic execution, we use our fork of the open-source symbolic executor Klee [1]. The fork has several modifications compared to the mainstream Klee, where the most important are:

1. support of symbolic-sized allocations via segment-offset pointer representation, and
2. the ability to replay a test-case from the sliced bitcode on the unsliced bitcode.

Also, we use Z3 [4] as the SMT solver. This is mainly because STP [6], the default solver used by Klee, does not support some features needed by our symbolic-size allocations extension.

After compilation of the source code into the bitcode, we perform some extra steps that are not indicated in Figure 1. We replace Test-Comp functions `__VERIFIER_nondet_*` with the corresponding built-in functions of Klee. Further, we optimize the code using the standard llvm optimization passes. These optimizations can significantly reduce the number of program instructions and thus speed up the remaining steps of the process. We also detect trivial infinite loops, i.e., loops corresponding to

```
while (1) {}
```

in C and replace them with calls to `abort`. The replacement prevents symbolic execution from infinite iteration over such loops. If slicing is employed, we apply one more step before it: we replace non-trivial infinite loops corresponding to

```
while (1) {/* loop body */}
```

with loops corresponding to

```
int x = 1;
while (x) {/* loop body */}
```

that have a formal exit node. The existence of loop exit nodes is important for the computation of control dependencies during program slicing [5]. If slicing is employed, the optimization passes and the replacement of trivial loops are applied once again on the sliced code.

Before giving the bitcode to Klee, we also replace undefined functions with symbolic stubs and make external globals internal (and initialize them to be symbolic).

After the symbolic execution finishes, we generate the `metadata.xml` file and copy the generated tests into the final destination (`test-suite` directory).

3 Strengths and Weaknesses

Although symbolic execution is very good in generating test-cases, it is computationally expensive. In particular, it struggles with programs that contain many branching instructions or loops with the number of iterations dependend on the input. Symbolic execution can fork when processing a branching instruction or such a loop header, and the number of considered execution paths may grow rapidly. This phenomenon is known as *path explosion*.

In Test-Comp, our weakest point appeared to be the test-generation from the results of the symbolic execution. We currently generate the test suite *after* the symbolic execution finishes, therefore on crash, out-of-memory or timeout, we usually failed to generate any test although the symbolic execution generated some tests internally. Further, we have generated plenty of test-cases that the test validator was not able to process (it seems that some input values are missing in these test-cases). We plan to fix both these issues in the next version of Symbiotic.

3.1 Comparison to the Mainstream KLEE

The mainstream KLEE also participated in TEST-COMP this year, therefore it is interesting to compare the results of SYMBIOTIC with "pure" KLEE.

One of the main differences in the results of these tools is that the mainstream KLEE usually scored some points when it timeout-ed or crashed, whereas we scored 0 points in such cases (as explained above).

For `coverage-error-call` property, there should be differences between our tool and the mainstream KLEE as we use slicing. There are some benchmarks where this is true, however, not many. This is probably because the benchmark set consists of rather simple programs where the error is easy to find even without slicing.

4 Tool Setup and Configuration

- *Download:* `https://gitlab.com/sosy-lab/test-comp/archives-2019/raw/master/2019/symbiotic.zip`
- *Participation Statement:* SYMBIOTIC 6 participates in all categories.
- *Execution:* Run `bin/symbiotic OPTS <source>`, where available `OPTS` include:
  `--test-comp` which sets the TEST-COMP configuration,
  `--prp=file` which sets the property specification file to use,
  `--32` which sets the 32-bit environment,
  `--help` which shows the full list of possible options.

The generated test-cases are stored in the directory `test-suite`.

5 Software Project and Contributors

SYMBIOTIC up to version 6 has been developed mainly by Jiri Slaby, Marek Trtík, Marek Chalupa, Martina Vitovská, Tomáš Jašek, and Michael Šimáček under the supervision of Jan Strejček. The tool and its components are available under MIT License. The project is hosted by the Faculty of Informatics, Masaryk University. LLVM, KLEE, and Z3 are also available under open-source licenses. The project web page is:

`https://github.com/staticafi/symbiotic`

Acknowledgments

The research is supported by The Czech Science Foundation grant GA18-02177S.

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, J. Strejček, and M. Vitovská. Joint forces for memory safety checking revisited. *To appear in STTT*, 2019.
3. M. Chalupa, M. Vitovská, and J. Strejček. Symbiotic 5: Boosted instrumentation - (competition contribution). In D. Beyer and M. Huisman, editors, *TACAS*, volume 10806 of *LNCS*, pages 442–446. Springer, 2018.
4. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340. Springer-Verlag, 2008.
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. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, volume 4590 of *LNCS*, pages 519–531. Springer, 2007.
7. S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.
8. J. C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.
9. 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.
10. M. Vitovská, M. Chalupa, and J. Strejček. SBT-instrumentation: A tool for configurable instrumentation of LLVM bitcode. *CoRR*, abs/1810.12617, 2018.
11. M. Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE, 1981.
12. LLVM. `http://llvm.org/`.