Program Analysis Benchmarks Submitted to the Model Counting Competition MC 2020

Sibylle Möhle
Johannes Kepler University Linz
Linz, Austria

Cunjing Ge
Johannes Kepler University Linz
Linz, Austria

Armin Biere
Johannes Kepler University Linz
Linz, Austria

I. Symbolic Execution

Symbolic execution is a program analysis technique that systematically explores program execution paths by using symbolic inputs instead of concrete data. For each execution path, it is possible to construct a path condition (PC) using symbolic condition. The path condition is satisfiable if and only if the path is executable. Thus satisfiability solvers can be used to generate input data for exercising the program paths and traversing the flow graph. This is an essential task in software testing and static analysis (for bug finding). A further question is: How often can the path be executed, given a random input? This is the path execution frequency problem. And it can be solved with counting tools.

II. Benchmark Generation

The benchmarks are generated by a symbolic execution bug finding tool called CAnalyze [1] with default settings. We analyzed 7 different programs: 'cubature' (an adaptive multi-dimensional integration program), 'gjk' (Gilbert-Johnson-Keerthi collision detection), 'http-parser' (an HTTP message parser), 'muFFT' (fast Fourier transforming), 'SimpleXML'(a light weight XML parser), 'tcas' (traffic collision avoidance system) and 'timeout' (tickless hierarchical timing wheel). They are ranging from 0.4k to 7.7k lines of source code. The original output of CAnalyze is SMT(BV) language. We then translate them into CNF using Boolector 3.2.0 [2].

Different formulae might represent the same path conditions, and their CNF representations then coincide. We removed duplicates and discarded all benchmarks which were solved by sharpSAT [3] within less than one second. Some of the benchmarks on which sharpSAT timed out were retained in the selection.

References

[1] Z. Xu, J. Zhang, Z. Xu, and J. Wang, "Canalyze: a static bug-finding tool for C programs," in *ISSTA*. ACM, 2014, pp. 425–428.

[2] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0 system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 53–58, 2014 (published 2015).

[3] M. Thurley, "sharpSAT – counting models with advanced component caching and implicit BCP," in *SAT*, ser. Lecture Notes in Computer Science, vol. 4121. Springer, 2006, pp. 424–429.

Supported by the LIT Secure and Correct Systems Lab and LIT AI Lab funded by the State of Upper Austria.

https://doi.org/10.35011/fmvtr.2021-1
Technical Report 21/1, January 2021, FMV Reports Series
Institute for Formal Models and Verification, Johannes Kepler University Linz
Altenbergerstr. 69, 4040 Linz, Austria