Profiling Symbolic Execution

Master's Thesis Project Description

Linard Arquint
arquintl@student.ethz.ch

Supervised by Prof. Dr. Peter Müller
Dr. Malte Schwerhoff

Department of Computer Science
ETH Zürich

April 23, 2019

1 Introduction

Symbolic execution [8] is a program analysis technique and was proposed in the past for several different applications, such as test case generation [5] or taint analysis [9]. An input program is not executed with concrete input values but symbolic ones are used instead. By step-wise executing the program, constraints on symbolic values are tracked. An important component of most symbolic execution frameworks is the SAT modulo theory (SMT) solver. A formula is given to the SMT solver, which can either prove its satisfiability, give a counterexample, which violates it, or respond with unknown. Symbolic execution explores a program path-wise and queries the SMT solver with many (and ideally comparably simple) formulas along the way. Therefore, symbolic execution typically queries the SMT solver often, but with relatively simple formulas, in contrast to e.g. tools based on weakest precondition calculi.

1.1 Problem Statement

Several symbolic execution frameworks have been proposed in the past. EXE [5], its successor KLEE [4], Rosette [11], or Silicon [10] are just a few examples. Although similarities exist, e.g. that they explore paths, there are

neither standard approaches nor common tools for debugging or profiling. Therefore, finding optimization targets is challenging and requires profound knowledge, e.g. about how the symbolic execution engine represents the program state or how it interacts with the SMT solver.

Due to the huge number of program paths that are explored during a symbolic execution, analyzing on which path most time was spent is already a tricky question, especially without tool support. Having a profiler that analyzes and visualizes the operations of a symbolic execution framework would therefore be of great use. We see potential for a profiler for Silicon, which facilitates the identification of performance problems in Silicon. It is mainly targeted at maintainers of Silicon to get insights into the operations that occur while symbolically executing a specific input program. Based on similarities between different symbolic execution frameworks, we estimate that some profiling concepts might also be applicable to other frameworks.

1.2 Terminology

In this project, we refer to a symbolic execution framework as the entire verification software including the SMT solver. The symbolic execution engine is the entire framework except the SMT solver.

2 Core Goals

A profiler for the Silicon symbolic execution framework will be developed, which helps maintainers to identify culprits causing performance problems. Earlier work on a logging infrastructre for Silicon [3] as well as on the Z3 axiom profiler [2] will be used as inspiration, because some aspects are similar, e.g. visualization of information on program paths.

Profiler Architecture. The implementation is split into layers, starting with profiling high-level aspects of any symbolic execution framework and becoming more specific to the used algorithms and techniques of Silicon. On each layer and based on sample programs, profiling recipes and analysis techniques will be derived and implemented. The implementation should allow *easy repetition* of the profiling. Furthermore, it should allow *visualization* and *analysis* of the collected data. This can happen either directly following the symbolic execution of an input program or by storing the data in a suitable format for later use. Hence, the *data format* represents an important aspect of this thesis and also determines how easily the profiler could be used by another symbolic execution framework.

Symbolic Execution Engine vs. SMT solver. Several papers [1, 4–6] have reported that the SMT solver significantly influences the execution time and is, despite recent advances, still the limiting factor. Therefore, several optimizations proposed in literature try to speed up queries to the SMT solver by e.g. caching or shrinking them [4,5]. Hence, the top most layer focuses on the interaction between the symbolic execution engine and the SMT solver. Not only the *total path execution time* of the SMT solver, but also the *number of queries* and the *execution time distribution* of queries are of interest. Profiling recipes will be developed and implemented in this layer so that the profiler will be able to answer these questions.

Analysis of Sample Programs. After implementing the first layer, an analysis of sample programs is performed. The analysis determines whether the symbolic execution engine or the SMT solver should be the focus of the second layer. Furthermore, the sample programs will be *grouped* according to the analysis results.

Time Distribution among Algorithms. In the second layer, the *different algorithms* used in either the engine or the SMT solver will be profiled.

For the engine, algorithms for branching, state modification, or state merging are potential candidates. Similarly for the SMT solver, the algorithms used to handle quantifiers, non-linear arithmetic, or lemma learning might be interesting. The profiler will be extended to *collect* and *analyze data* in order to decide which part looks most promising for further investigation.

In-Depth Analysis. The runtime of the specific operations of an algorithm will be profiled in the lowest layer. At this point, it is not clear yet whether it will be a specific algorithm, e.g. state merging, or if the analysis will be concerned with specific SMT encoding.

The developed recipes of the first two layers might be generic enough to be of use for other symbolic execution frameworks as well (see section 3). However, this layer will most likely be specific to Silicon and its implementation.

To illustrate above goals, figure 1 shows two execution paths, namely q_1 and q_2. Each path consists of several blocks, a, c_i, and d_i, that represent executions of specific algorithms, e.g. for state consolidation. A block in turn comprises several queries to the SMT solver, that were issued during the execution of the represented algorithm.

The profiler will not only return the total execution time of an entire path q_1, but also answer questions about individual blocks on a path.

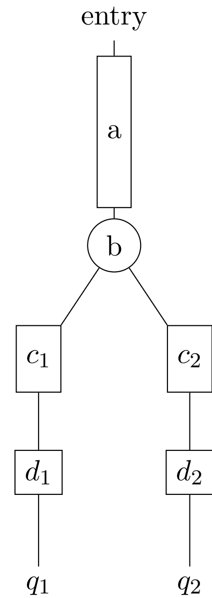

Figure 1

3 Extension Goals

- The profiler could be extended to give insights into these additional properties:

  Similarities between Queries. While symbolically executing a certain path, Silicon not only extends the path constraint, but also modifies the heap representation of the executed program. We estimate that queries along a path might expose similarities, because only a small part of the symbolic execution state changes in every step. Hence, an analysis or graphical representation of the similarities and differences of queries could be helpful. For example, if two subsequent queries are almost identical but have a significantly different execution time, looking at the differing parts of the query could help to understand the execution times.

  Incurred SMT Solver Work. Building up on the redundancy between SMT solver queries, the profiler could also analyze the statistics of the SMT solver (e.g. Microsoft Research's Z3 [7]) over time on a path, as an attempt to better understand the work that is going on in the solver itself. As an example, if the SMT solver has a metric for work using non-linear arithmetic, an increase of this metric would indicate that the queries contain non-linear constraints or at least require solving non-linear arithmetic operations.

- As mentioned, the profiler might not be restricted to the operations of Silicon but some aspects might be applicable to other symbolic execution frameworks as well. Therefore, an extension goal could be to extend a second framework to produce and store data about its symbolic execution in the same data format as expected by the developed profiler. This would allow the profiler to visualize the execution of the other framework.

References

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, May 2018.

[2] N. Becker, P. Müller, and A. J. Summers. The Axiom Profiler: Understanding and Debugging SMT Quantifier Instantiations. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS. Springer-Verlag, 2019. To appear.

[3] Andreas Buob. Recording Symbolic Execution, 2015.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[5] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[6] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.

[9] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP

'10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[10] Malte Hermann Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution.* PhD thesis, ETH Zurich, 2016.

[11] Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 530–541, New York, NY, USA, 2014. ACM.