Symbolic execution - model equivalence & applications

Matei Popovici, Radu Stoenescu, Lorina Negreanu, Costin Raiciu
University Politehnica of Bucharest

1. NETWORK SYMBOLIC EXECUTION

Symbolic execution runs programs with *symbolic* inputs instead of concrete ones. A symbolic input models a range of values, which may be constrained or modified during program execution. The output of symbolic execution is the set of all possible program execution paths, and for each path and variable v — the *symbolic expression* to which v is bound, i.e. the set of constraints on v on the path at hand.

Consider an example where the integer `a` is declared symbolic, meaning it can take any integer value, and a program that executes `if (a>0)`: the execution engine will explore two paths after the instruction, one where the constraint `a>0` holds, and one where its negation `a<=0` holds.

Symbolic execution has been recently used to verify network dataplanes [1, 3, 2]. When applied to the dataplane code (e.g. middlebox C code), it can capture a series of interesting bugs including low level memory access errors [1], but it does not scale very well, with verification times in the order of minutes to hours per box.

Thus, all symbolic execution papers including [1, 3, 2] use models of the code instead of real code to perform symbolic execution in reasonable time; these models work at various abstraction levels and are coded in domain-specific languages (e.g. SEFL [3]) or in C code. By optimizing models for fast symbolic execution, it is possible to test fairly large networks in tens of seconds [3, 2].

There is one downside: existing works offer no strong guarantees that the model is a faithful representation of the real code. We present a possible solution to this problem.

2. MODEL EQUIVALENCE

There is a fundamental tension between the runtime speed and the symbolic execution speed of a program [4].When analyzing a simple switch model, symbolic execution can take orders of magnitude less time when the code has been optimized for symbolic execution. However, executing that same code in practice will result in very poor performance [3]. Many researchers have observed this and produced optimized models that enable symbolic execution; unfortunately, there is gap between the model and the actual code.

We take a principled approach to understand what transformations are correct and safe when optimizing models for symbolic execution. Intuitively, we want the optimized model to behave in the same way as the original code.

One approach for checking model equivalence is to look at the domain sets for each program variable. Suppose symbolic execution for model M yields n paths and on each path i, the variable v is bound to symbolic expression e_i. The *domain set* for variable v is thus $d_{M_1} = e_1 \vee e_2 \vee \ldots \vee e_n$.

Two models M_1, M_2 are *output-equivalent* iff the domain sets for all variables are equivalent ($d_{M_1} \wedge \neg d_{M_2} = \mathit{false}$).

Output equivalence is limited in capturing input-output dependence. For instance, programs `if (x > 0) x = 1; else x = 0` and `if (x > 0) x = 0; else x = 1;` are output equivalent, but do not behave in the same way.

A more conservative alternative is to define equivalence in terms of execution paths: M_1, M_2 are equivalent if each path from M_1 is equivalent to some path from M_2 and vice-versa.

However, this approach leaves little room for symbolic execution optimizations which may target path reduction.

A tradeoff between the two is to consider *state-dependent equivalence*. Say P is a *state-predicate* (e.g. $P \equiv$ `x != 0`). Two models are P-equivalent if they produce precisely the same output on any input that satisfies P. State predicates offer flexibility in deciding how strong requirements we place on model equivalence.

3. IMPLEMENTATION

In order to optimize models for symbolic execution, we trade off between: (i) the number of execution paths and (ii) the number of constraints per path, for each variable.

Optimization consists in a sequence of equivalence-preserving transformations similar to compiler code optimizations.

First, we perform general optimizations e.g. removing conditionals which result in only one successful path, removing dead code, merging consecutive variable constraints in a single one (thus having fewer invocations on the constraint solver). For some transformations, we may only preserve equivalence with respect to specific state predicates.

Second, we apply transformations aimed at optimizing for criteria (i) or (ii). For instance, in `if (x > 0)` p_1 `else` p_2, if we can determine that `x > 0` holds (universally, or w.r.t. our given state predicate), we can transform the program to `assert (x > 0);` p_1, thus reducing program branching.

4. REFERENCES

[1] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[2] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. Buzz: Testing context-dependent policies in stateful networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 275–289, Santa Clara, CA, Mar. 2016. USENIX Association.

[3] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. SymNet: scalable symbolic execution for modern networks. `http://arxiv.org/abs/1604.02847`, 2016.

[4] J. Wagner, V. Kuznetsov, and G. Candea. Overify: Optimizing programs for fast verification. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 18–18, Berkeley, CA, USA, 2013. USENIX Association.