A Recursive Strategy for Symbolic Execution Expressed in Coq

Alyssa Byrnes

October 8, 2018

Abstract

Symbolic execution allows one to execute a program with symbolic inputs, rather than concrete ones, exploring multiple execution paths simultaneously. This can be a useful tool in debugging a system, for now we can potentially execute all possible inputs of a program to discover bugs and generate exploits. In this work, we examine a previously proposed hardware-oriented recursive symbolic execution algorithm and formally verify that if an error state is reachable, it produces a sequence of inputs that will take the processor from its inital state to that error state.

1 Data Types

- Object of type φ to represent nodes on the symbolic execution tree.
  - get_phi(n) returns ϕ, the abstract state.
  - get_pc(n) returns π, the path constraint.
  - B = sym_ex(A), or $[A]_\sim \Rightarrow_{\mathcal{S}}^{S^*} [B]_\sim$, represents symbolic execution of state A for one clock cycle, where A and B are of type φ.
  - unif(A), or $[\![A]\!]$ represents the set of concrete states represented by symbolic state A.
- Object of type $\mathcal{E}$ to represent a symbolic execution tree made of objects of type n.
  - is_leaf($\mathcal{E}$, n) returns *true* if n is a leaf in $\mathcal{E}$.
  - is_root($\mathcal{E}$, n) returns *true* if n is the root in $\mathcal{E}$.
  - get_root($\mathcal{E}$) returns object of type n that is the root of the tree.
- Object of type γ to represent concrete state.
  - B = conc_ex(A), or $A \Rightarrow_{\mathcal{S}}^* B$, represents concrete execution of state A for one clock cycle, where A and B are of type γ.
- Object E to represent set of concrete states.
  - Contains set of initial configuration states $\mathcal{T}_{Cfg}$.

Shorthand:

- $\bar{s}_{r,m}$ = get_phi(get_root($\mathcal{E}_m$))
- $\pi_{r,m}$ = get_pc(get_root($\mathcal{E}_m$))
- $\bar{s}_{l,m}$ = get_phi($n_{l,m}$), where $n_{l,m} \in \mathcal{E}_m$.
- $\pi_{l,m}$ = get_pc($n_{l,m}$), where $n_{l,m} \in \mathcal{E}_m$.

2 Accepted Knowledge

These are the properties outlined in Arusoaie et al.'s paper [?].

Lemma 1 *If $\gamma \Rightarrow_{\mathcal{S}} \gamma'$, and $\gamma \in [\![\varphi]\!]$, then there exists φ' such that $\gamma' \in [\![\varphi']\!]$ and $[\varphi]_{\sim} \Rightarrow^{S}_{\mathcal{S}} [\varphi']_{\sim}$. [?]*

Corollary 1 *For every concrete execution $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \ldots \gamma_n \Rightarrow_{\mathcal{S}} \ldots$, and pattern φ_0 such that $\gamma_0 \in [\![\varphi_0]\!]$, there exists a symbolic execution $[\varphi_0]_{\sim} \Rightarrow^{S}_{\mathcal{S}} [\varphi_1]_{\sim} \Rightarrow^{S}_{\mathcal{S}} \ldots [\varphi_n]_{\sim} \Rightarrow^{S}_{\mathcal{S}} \ldots$ such that $\gamma_i \in [\![\varphi_i]\!]$ for all $i = 0, 1, \ldots$. [?]*

Lemma 2 *If $\gamma' \in [\![\varphi']\!]$ and $[\varphi]_{\sim} \Rightarrow^{S}_{\mathcal{S}} [\varphi']_{\sim}$ then there exists $\gamma \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\varphi]\!]$. [?]*

Corollary 2 *For every feasible symbolic execution $[\varphi_0]_{\sim} \Rightarrow^{S}_{\mathcal{S}} [\varphi_1]_{\sim} \Rightarrow^{S}_{\mathcal{S}} \ldots [\varphi_n]_{\sim} \Rightarrow^{S}_{\mathcal{S}} \ldots$ there is a concrete execution $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \ldots \gamma_n \Rightarrow_{\mathcal{S}} \ldots$ such that $\gamma_i \in [\![\varphi_i]\!]$ for all $i = 0, 1, \ldots$. [?]*

Lemma 1 states that all concrete states have corresponding symbolic representations, and Lemma 2 states that if a program symbolically executes to a set of possible concrete assignments, then initial concrete assignments exist so the program can concretely execute to that state.

Corollaries 1 and 2 lift this definition to a consecutive sequence of executions. From here on, a sequence symbolic executions of an unbounded, finite length, $[\varphi_0]_{\sim} \Rightarrow^{S}_{\mathcal{S}} [\varphi_1]_{\sim} \Rightarrow^{S}_{\mathcal{S}} \ldots [\varphi_n]_{\sim}$ will be denoted by $[\varphi_0]_{\sim} \Rightarrow^{S^*}_{\mathcal{S}} [\varphi_n]_{\sim}$. A sequence of concrete executions of an unbounded, finite length, $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \ldots \gamma_n \Rightarrow_{\mathcal{S}}$ will be denoted by $\gamma_0 \Rightarrow^{*}_{\mathcal{S}} \gamma_n$

3 Circle Operations

These use the notation defined in Arusoaie et al.'s paper [?].

Definition 1 *circle_op_1 = the set $\gamma \in [\![\varphi]\!]$ $\forall$ $\gamma' \in [\![\varphi']\!]$ of a given φ' where $[\varphi]_{\sim} \Rightarrow^{S^*}_{\mathcal{S}} [\varphi']_{\sim}$ such that $\gamma \Rightarrow^{*}_{\mathcal{S}} \gamma'$ and is_leaf(φ')= true.*

Definition 2 *circle_op_2 = the set $\gamma' \in [\![\varphi']\!]$ $\forall$ $\gamma \in [\![\varphi]\!]$ of a given φ where $[\varphi]_{\sim} \Rightarrow^{S}_{\mathcal{S}} [\varphi']_{\sim}$ such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and is_leaf(φ')= true.*

circle_op_1 represents all concrete states that will take us down exactly one path in the symbolic execution tree. circle_op_2 represents all concrete output states of a given path in the symbolic execution tree.

4 Properties

For a given E, X = a sequence $\mathcal{E}_0, ..., \mathcal{E}_m$ such that $\forall \mathcal{E}_x$, $\exists n_{l,x}$ such that the conjunction of the following is true:

1. $s_0 \in$ circle_op_1($\bar{s}_{r,0}$, $\pi_{l,0}$)
2. $E \bigcap$ circle_op_2($\bar{s}_{l,m}$, $\pi_{l,m}$) $\neq \{\}$
3. for $j = \{0, ..., m-1\}$, circle_op_2($\bar{s}_{l,j}$, $\pi_{l,j}$) $\subseteq$ circle_op_1($\bar{s}_{r,j+1}$, $\pi_{l,j+1}$)
4. is_leaf($\mathcal{E}_x$, $n_{l,x}$) = *true*.

5 To Do

- Reason about equivalence class, and how the notation transfers to our notation.
- Representing unification in Coq.
- Representing sequences in Coq.
- Representing sufficiency requirement in Coq.
- Representing "equivalence" of concrete states.

6 Notes for Consideration

- We need to consider uniqueness. This might come naturally from the overall tree structure.
- Our work assumes SAT-solver correctness.
- Our circle_op returns concrete states, so we just need to be able to compare concrete states.