A Recursive Strategy for Symbolic Execution Expressed in Coq

Alyssa Byrnes

October 25, 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 Abstract Variables and Methods

1.1 Concrete Execution

Abstract Variables

- input - string of inputs
- state - system state

Inductive-type Variables

- input_list - list of elements of type "input"
- conc_state - constructs a concrete state given a system state and an input, (elements of type "state" and "input".)

Abstract Methods

- conc_ex - inputs elements of type "conc_state" and and "input" and outputs new "conc_state".

1.2 Symbolic Execution

Abstract Variables

- phi - Abstract State
- pc - path constraint

Inductive-type Variables

- sym_state - constructs a symbolic state given elements of type "phi" and "pc"
- SE_tree - tree structure consisting of sym_state elements
- SE_tree_list - list containing SE_tree elements

Abstract Methods

- get_phi - inputs sym_state and returns phi
- get_pc - inputs sym_state and returns pc
- concretize - takes a phi and pc and returns a concrete state
- get_input - takes a pc and outputs an element of type "input"

2 Definitions

2.1 Variables

- init_conc_states - an ensemble of conc_states.
- tree_list - an SE_tree_list

2.2 General Methods

Definition 1 *is_connected(tlist) := forall A B* $\in$*, tlist* $\geq 2 \rightarrow$ *if A and B are consecutive, then the root of B is a leaf of A.*

Definition 2 *execute_tree_list(tlist): steps through tlist, executing a list of size 2 (or the first two elements of a list) as conc_ex(conc_ex(init_conc_states, x), y) where x = get_input (get_pc (root(second_element))) and y = get_input (get_pc (root(third_element))). For any other element in the list, it recurses as conc_ex(execute_tree_list(tlist', z)), where tlist' = tlist with the last element removed, and z = get_input (get_pc (root(next_element))).*

2.3 Circle Operations

Circle_op_1 takes as input symbolic state of root and pc of its leaf and returns all and only the concrete states that will take us down the path that leads to the leaf.

Definition 3 *circle_op_1 (sym, sym_leaf) := concretize (get_phi sym) (get_pc sym_leaf).*

Circle_op_2 takes as input symbolic state of leaf state and pc of leaf state and returns concrete states that correspond.

Definition 4 *circle_op_2 (sym) := concretize (get_phi sym) (get_pc sym).*

3 Assumptions

Property 1 *Property 1 : init_conc_states* $\in$ *circle_op_1 (root(head(tree_list)), root (second_elem (tree_list))).*

Property 2 *Property 2 : circle_op_2 (root (last_elem(tree_list)))* $\bigcap$ *ErrorStates* $\neq$ *Empty_set.*

Property 3 *Property 3 : is_connected tree_list.*

Property 4 *tl_size : size(tree_list)* ≥ 2*.*

Property 5 *base_case: forall s0 s : SE_tree, is_connected (([ ] :: s0) :: s) -> conc_ex (conc_ex init_conc_states (get_input (get_pc (root s0)))) (get_input (get_pc (root s))) = circle_op_2 (root s).*

Property 6 *co_2_def: forall (s0 : SE_tree) (s: sym_state), is_leaf_state s0 s -> conc_ex (circle_op_2 (root s0)) (get_input (get_pc s)) = (circle_op_2 s).*

4 Theorem

Theorem 1 *Theorem sufficiency :*
(execute_tree_list tree_list) $\bigcap$ *ErrorStates)* $\neq$ *Empty_set.*

5 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.