Symbolic Execution Engine Guide

Students Ernesto Ortiz, Brittany Boles, Garrett Perkins, Zach Wadhams
Professor Dr. Matthew Revelle
Date November 5, 2024

Architecture Overview

- `C_LANG = Language(tsc.language())`: Initializes the Tree-sitter language parser for C code.
- `parser = Parser(C_LANG)`: Creates a parser object for parsing C code.
- `read_c_code_from_file(file_path)`: Reads the contents of a C code file as bytes, given the file path.
- `print_tree(node, indent="")`: Recursively prints the syntax tree of the C code, showing each node type with indentation for hierarchy.
- `SymbolicExecutor` (Class): Finds the starting node of the function we want to execute, symbolically traverse through this node, creating mappings of variables to symbolic values and keeping track of accrued conditions. When branching conditions are found we check for feasibility using z3 solver and add branches to to our stack of states. When return statements are found we check for 1s or 0s and if 1 is returned we save the condition that led us there.
- `main()`:
  - Sets up file paths and reads in the C code.
  - Parses the C code using Tree-sitter and generates a syntax tree.
  - Creates an instance of `SymbolicExecutor` and executes a specified function from the parsed code.
  - Outputs the total feasible, infeasible, and target-reached states upon execution.

Implementation Documentation

0.1 Class SymbolicExecutor

0.1.1 init

This class has the following instance variables.

- `functions` - Dictionary to store function names and nodes. We use this to get the starting point of what ever function is given to us to symbolically execute. Example: {f: f node, test: test node}
- `mapping` - Maps variable names to their symbolic variable name. Example: {x:[X1,X3], y: [X0], i:[x2]}
- `counter` Counter for generating unique variable identifiers.
- `SAT` - Counter to track feasible paths (satisfiable paths).
- `UnSAT` - Counter to track infeasible paths (unsatisfiable paths).
- `targetReached` - Counter for paths that meet the target condition ‘return 1;‘.
- `solver` - The z3 solver that will hold the constraints as we traverse the tree and solve for concrete values once we reach the target.
- `while_condition` - Helper boolean to track feasibility of while loops.

0.2 execute

To start to call the Execute method. This is given a parsed tree of a c file and a function name to symbolically execute. This calls find_function which gets a list of all the function names and their starting node. We then given a function we want to execute, we match it's name against the function dictionary and return the start node. Finally we call the function tree_traverser, and traverse that start node.

0.3 find_function

This takes in the parsed_tree and recursively traverses through each node started at the top and goes until we have no new nodes. If the node.type is "function_definition", then we parse through the tree to get the name and add the name and start node to the function dictionary.

0.4 traverse_node

The `traverse_node` method is responsible for recursively traversing each node in the function's syntax tree and delegating handling of specific node types to helper functions based on the node's functionality.

This method first checks the type of the node and performs appropriate operations based on the type:

- `update_expression` - Handles increment (++) and decrement (--) expressions for variables, updating the `condition` list with the corresponding symbolic representation.
- `declaration` - Calls the `handle_declaration` function for initializing a new variable.
- `assignment_expression` - Calls `handle_assignment` to update variable values.
- `if_statement` - Calls `handle_if_statement` to process conditional branches. Within the if handler we traverse inside the if statement, so after handling we move to the end of the if statement (next_node)
- `while_statement` - Calls `handle_while_statement` to process loop statements. Within the while handler we traverse inside the while statement, so after handling we move to the end of the while statement (next_node)
- `return_statement` - Calls `handle_return` to check the return condition.

The method then recursively traverses the children of each node to ensure all nodes are processed, passing unhandled node types to the next level for further traversal.

0.5 get_concrete_value

This method calls on the z3 solver to get concrete values for the variables given the stored constraints. The function is called when the 'return 1' target is reached.

0.6 z3_condition_add

This method applies basic operators in Z3 to form a symbolic expression, which is essential for evaluating conditions in branches and loops. It adds boolean and arithmetic operations to the Z3 solver:

- Boolean operators like ==, <, >, etc., can be inverted to handle the "else" conditions.
- Arithmetic operators like ++ and -- are handled by incrementing or decrementing the symbolic variable within Z3.

0.7 else_constraints

This method add the symbolic variables and the constraints when an else condition is reached.

0.8 check_feasibility

The `check_feasibility` method uses the Z3 solver to verify the feasibility of a path condition, integrating the stored conditions into the solver to assess whether the path is `SAT` (feasible) or `UNSAT` (infeasible). It takes in the branching condition (if or while condition) and then initializes all the symbolic variables in the solver. It then adds all the conditions including the new one into the solver via the z3_condition_add function. Finally it checks if this is feasible.

This method ensures only feasible paths are pursued during symbolic execution and discards infeasible paths.

0.9 handle_declaration

The `handle_declaration` function initializes new variables by generating unique symbolic identifiers using `counter`. This identifier is mapped to the variable name in `mapping` and represents each variable uniquely during symbolic execution. The method can handle both simple variable declarations (int x), function call expressions within declarations (int y = f()) and deceleration with assignment (int x = 10).

Each new variable assignment is appended to the `mapping` stack, and we ensuring that only the latest assignments are used in the path conditions.

0.10 increment_assignment

This method is a helper for updating pre/post increments/decrements.

0.11 handle_assignment

`handle_assignment` updates an existing variable's value in mapping by adding a new symbolic var so x:[X1] becomes x:[X2]. We also add the new symbloic vars conditions to the "conditions". If a variable has not been previously declared, the method prints an error message. This function is used to maintain a consistent symbolic representation of each variable's state throughout execution.

0.12 handle_if_statement

The `handle_if_statement` method creates conditional branches by evaluating the feasibility of the `if` conditions if or else which have true branch or false branch statements. If either condition (the if or else) is feasible (`SAT`), it traverses the branch and saves the current state to allow future restoration to test other possible paths.

If an `else` statement exists, it checks the feasibility of the inverse of the `if` condition, and, if feasible, traverses this path, allowing both branches to be evaluated for full path coverage. However sometimes there isn't a else statement with an if, so we simply move on in that case.

0.13 handle_while_statement

The `handle_while_statement` method processes loop conditions by repeatedly checking feasibility and traversing the loop body as long as the condition holds. It saves the state before entering the loop to allow restoration if the loop condition becomes infeasible, preventing infinite loops when an unsatisfiable condition is encountered.

0.14 handle_return

This method processes return statements by evaluating if they match a target condition, such as `return 1;`. When this condition is reached, the `targetReached` counter is incremented.

Setup and running instructions

The assignment 3 GitHub is located here: https://github.com/zwadhams/AutoVulnDiscovery/tree/main/Assignment3. Once downloaded, the `Symbolic_Ex.py` program can be executed with a command line argument of a C program. Ex: {`python3 Symbolic_Ex.py test_program.c test > assignment.log`} All of our testing was done with the included `test_program.c` file.