WIMFGSymex

Nitasha Fazal Mike Kadoshnikov Layton McCafferty
Dillon Shaffer Jacob Tanner

October 2024

1 Introduction

The WIMFGSymex project implements a symbolic execution engine for analyzing C programs to detect feasible execution paths and identify specific target conditions. By symbolically interpreting variables and expressions, this engine can simulate various code paths and determine the feasibility of conditional statements, loops, and assignments without actually running the code.

Our implementation uses Python's tree-sitter library to parse C code into an abstract syntax tree (AST), and the Z3 solver for Satisfiability Modulo Theories (SMT) to evaluate path conditions. This combination enables us to explore paths systematically, storing variable states symbolically, and verifying whether certain program states or outputs are achievable under specified conditions.

This document outlines the structure of our implementation, from the setup and dependencies to the architecture and detailed functionality of key classes like SymbolicState and SymbolicExecutionEngine. Additionally, we provide code snippets illustrating our engine's use and discuss sample output that demonstrates its ability to evaluate complex C functions symbolically.

2 Project Set Up

We implemented the project in Python, so Python 3.11 or later is required. The next step is to install the necessary libraries which are Z3 and tree-sitter. We also used numpy for our project. We used the following command to install these libraries on our systems:

```
pip install z3-solver tree-sitter tree-sitter-c numpy
```

Note: If environment is being used to mange python libraries, make sure to run these commands inside the environment.

2.1 Packages

We implemented this program in Python, using tree-sitter and tree-sitter-c to convert C code into an Abstract Syntax Tree (AST). Additionally, we used the Z3 Python library as the Satisfiability Modulo Theories (SMT) solver.

2.2 Running

Our tool can be run either as a python library pulled in by other software or as a CLI. To run it as a CLI, use the following command:

```
$ python SymbolicExecutionEngine.py [file.c] [function-name]
```

*Note: ensure that the **pip install** command has been run to ensure you have the correct packages.*

3 Architecture

We used a class-based software design, with two main classes:

- SymbolicState: This class encapsulates variables, their values, and conditions. It has three primary attributes: variables, values, and conditions. An instance of this class is passed to the SMT solver whenever conditions in the source code need to be evaluated, allowing the solver to determine the feasibility of different execution paths by analyzing the values of variables within these conditions.

- SymbolicExecutionEngine: This is the primary symbolic execution interpreter tasked with parsing and symbolically executing statements within the AST of a function. It handles:

  - Variable declarations (including from function parameters and variable assignments)
  - Control structures (if-else and while loops)
  - Return statements

  This class is also capable of symbolically interpreting these expressions:

  - Arithmetic expressions and integer literals/variables.
  - Operators such as addition, subtraction, multiplication, division, and negation.
  - Increment and decrement operations (both pre-form and post-form).
  - Boolean expressions including equality and inequality checks.

3.1 How it works

The SymbolicExecutionEngine starts by constructing the AST of the source code which represents elements like functions, statements, and expressions in a structured format. The engine then identifies the target function to analyze; this function acts as the entry point for symbolic execution.

While traversing the AST, the engine encounters various statements and handles them according to specific implementation rules. If concrete values are not provided for declaration statements, it initializes variables with symbolic values. Once the engine has initialized these values, it updates the symbolic state object with the current symbolic value of each variable.

When the engine encounters control flow or conditional statements, such as if-else or while loops, it creates two copies of the current state: one for the true path and another for the false path. It then passes the condition to the SMT solver to check feasibility. If the true path condition is feasible, the engine proceeds to execute the 'then' block. If the false path condition is feasible, the engine traverses the 'else' block instead. In a very similar way, the engine symbolically evaluates while loops by analyzing the loop condition and keeping track of possible variable changes in each iteration.

The next part of the code handles expressions and operations, including Boolean expressions used in conditions. As the engine traverses function nodes, it encounters arithmetic or Boolean expressions (e.g., a + b, x ¡ y) and evaluates them symbolically, preserving relationships between variables. Additionally, pre-increment/decrement and post-increment/decrement operations modify symbolic values within the symbolic state object.

Upon encountering a return statement, the engine checks the return value. If the return value is 1, it logs the current symbolic state. For a return value of 0, it negates the last condition. This handling of the return statement is specific to the requirements of this assignment. The engine then stores the final symbolic state, which is used to find concrete values for the variables to achieve the goal—specifically, reaching a return statement with a value of 1.

We used Z3, a Python SMT solver library, in our code. When a condition needs to be evaluated, the engine passes the collected path conditions to the SMT solver. The solver checks the feasibility of the path by determining whether the path conditions are logically consistent—i.e., whether an assignment of variable values exists that satisfies all conditions for a path. If the SMT solver finds a path condition to be infeasible, the engine skips further evaluation along that path.

4 Implementation

For this assignment. we used the same source code of the c program, to exercise our interpreter and verification of our engine's correctness.

```
#include <stdio.h>
#include <stdlib.h>

int y_init;
int f() {
  return y_init;
}

int test(int x) {
  int y = f();
  if (x > y) {
    int i = 0;
    while (i < 10) {
      x--;
      i++;
    }
    if (x == y) {
      puts("nope");
      return 0;
    }
    else {
      if (x < y) {
        // Get here
        puts("got it!");
        return 1;
      }
      else {
        puts("not here");
        return 0;
      }
    }
  }
  puts("wrong way");
  return 0;
}

int main(int argc, char *argv[]) {
  int x = atol(argv[1]);
  y_init = atol(argv[2]);
  int result = test(x);
  printf("result: %d\n", result);
  return 0;
}
```

As outlined in the architecture section, the SymbolicState class is responsible for encapsulating variables, their symbolic values, and associated path conditions. It is initialized as follows:

```
class SymbolicState:

    def __init__(self):
        # Store variable names as name -> type * value
        self.variables = {}
        # Stores concrete values
        self.values = {}
        # Stores path conditions for the state
        self.path_conditions = []


    def add_variable(self, name, var_type, value=None):
        self.variables[name] = var_type
        if value is None:
            value = z3.Int(name + "_0")

        self.values[name] = value
```

The *add_variable* function initializes variables with concrete values if they are provided; if not, it assigns symbolic variables. This ensures that variables are represented accurately based on available information, supporting both concrete and symbolic evaluation in the symbolic state.

The SymbolicExecutionEngine class is initialized to manage symbolic execution by parsing the source code into an AST, setting up a symbolic state, and integrating with an SMT solver. Here's how the initialization is structured.

```
class SymbolicExecutionEngine:

    def __init__(self, source_code, function_name):
        # parsing the source code
        C_LANG = Language(tree_sitter_c.language())
        self.parser = Parser(C_LANG)
        self.ast = self.parser.parse(source_code.encode())

        # attributes
        self.loop_states = []
        self.target_reached = False
        self.target_reached_conditions = []
        self.function_name = function_name
        self.state_stack = []
        self.target_reachable_state = None
        self.infeasible_paths = 0
        self.feasible_paths = 0
        self.conditions_safe = []
```

The other main functions in the implementation are the *handle_expressions* and *handle_statements*. Below is the output related to the test function from the above code.

```
Total Number of infeasible states: 1
Total Number of feasible states through function: 5
Number of feasible paths to reach target statement: 4
    Constraints to reach target statement:
        x_0 > y_0
        Not(10 > i)
        Not(x_0 == y_0)
        x_0 < y_0
Concrete values to reach target statement:
    variable x_0 | Concrete Value: 0
    variable y_0 | Concrete Value: -1
    variable i_0 | Concrete Value: i_0
```