SWE 681
Project report
Linhan Li

Path Discovery to Critical Regions via Symbolic Execution

This project focuses on developing a basic symbolic execution tool for the Python programming language. Our work is developed on top of the functionalities provided by the *ast* libraries of Python and the constraint solving implementation of *Z3*.

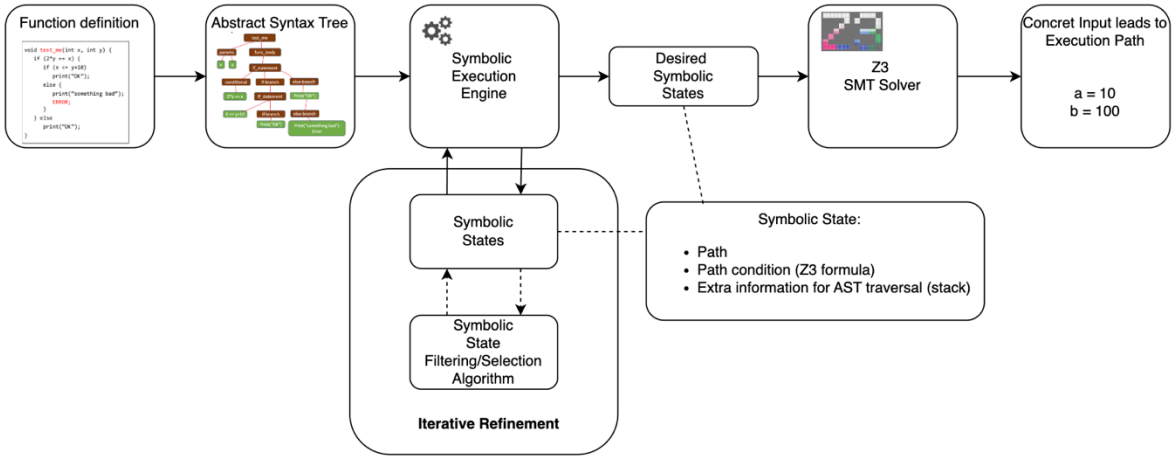

Figure 1: Overview of our tool

Overview

Our tool takes a python function snippet as its input, and perform symbolic execution on the input function to generate a set of symbolic states. Depending on the specific use case, these symbolic states can be used to generate more symbolic states, or they can be translated into z3 formulas for satisfiability checking.

Our tool first converts the input function into an Abstract Syntax Tree (AST) representation and traverses it in an organized manner that respects the semantics of the source code to generate symbolic states. This traversal can be viewed as an in-order traversal augmented with semantic information. Unlike regular tree traversal, where each node is visited only once, AST nodes—particularly those representing control flow constructs (e.g., if and while statements)—may be visited multiple times or skipped entirely. Furthermore, when the AST diverges, a single symbolic state can branch out, producing additional symbolic states. We carefully handled these complexities in our implementation.

Symbolic Execution

The symbolic execution is performed iteratively. In each iteration, we process all current symbolic states by executing the next line of code (corresponding to the next node in the AST), producing new symbolic states. These new states are then used to update the current set of

symbolic states. At the end of each iteration, we use Z3 to check the satisfiability of each state, filtering out those that can never be satisfied (i.e., no concrete input can lead to their corresponding execution paths). This approach improves the scalability of symbolic execution.

Tool Functionalities

Currently, our tool provides three main functionalities:

1. **Path exploration**: Finds paths to a specified location within the input function.
2. **Path enumeration**: Enumerates paths starting from a specified location, up to a scalable depth.
3. **Directed exploration**: Allows guided exploration under user supervision.

Supported Features and Assumptions

This tool only supports a small subset of the Python programming language, and makes the assumption that integer is the only datatype in the function. Below is a list of supported syntactical components:

1. While loop (with the keyword **continue** and **break**)
2. If conditional
3. Assertions
4. Variable assignment
5. Comparison expressions

Examples

A simple use case example is included in the appendix. The tool itself, along with additional examples, is available in the GitHub repository:
https://github.com/SWVM/681_prjt

Sources

AST: https://docs.python.org/3/library/ast.html
Z3: https://github.com/Z3Prover/z3?tab=readme-ov-file
Termcolor: https://github.com/termcolor/termcolor

Appendix:
Example 1:

- Input function

```python
def non_reachable(a,b):
    assert a<5
    while b>a:
        a = a + 1
        trace()
        if a > 15:
            target()
            return a
        else:
            continue
            return a


def target():
    print("Wow, target reached!!!")

couter = 0
def trace():
    global couter
    couter += 1
    print("looping... iter: ", couter)
```

We want to find assignment of function parameter **a** and **b** such that the function **target()** can be reached.

- Reaching symbolic state

```
Symbolic State
[a < 5, b > a, a_1 == a + 1, Not(a_1 > 15), b > a_1, a_2 == a_1 + 1, Not(a_2 > 15), b > a_2, a_3 == a_2 + 1, Not(a_3 > 15), b > a_3, a_4 == a_3 + 1, Not(a_4 > 15), b > a_4, a_5 ==
 a_4 + 1, Not(a_5 > 15), b > a_5, a_6 == a_5 + 1, Not(a_6 > 15), b > a_6, a_7 == a_6 + 1, Not(a_7 > 15), b > a_7, a_8 == a_7 + 1, Not(a_8 > 15), b > a_8, a_9 == a_8 + 1, Not(a_9 >
 15), b > a_9, a_10 == a_9 + 1, Not(a_10 > 15), b > a_10, a_11 == a_10 + 1, Not(a_11 > 15), b > a_11, a_12 == a_11 + 1, a_12 > 15]
```

- Path taken (loops omitted)

```
Path Taken
        (2)     Assert: a < 5
        (3)     While(Enter): b > a
        (4)     Assign: a = a + 1
        (5)     Func Call: trace
        (6)     If(else): a > 15
        (10)    Continue: continue
        (3)     While(Enter): b > a
        (4)     Assign: a = a + 1
        (5)     Func Call: trace
        (6)     If(else): a > 15
        (10)    Continue: continue
```

```
        (3)     While(Enter): b > a
        (4)     Assign: a = a + 1
        (5)     Func Call: trace
        (6)     If(else): a > 15
        (10)    Continue: continue
        (3)     While(Enter): b > a
        (4)     Assign: a = a + 1
        (5)     Func Call: trace
        (6)     If(if): a > 15
        (7)     Hit Target: target()
```

- Satisfying Assignment (provided by Z3)

```
Satisfying Assignment
{'b': 16, 'a': 4}
```

- Function execution output with Satisfying assignment

```
==============
Calling Func with z3 generated inputs
==============
looping... iter:  1
looping... iter:  2
looping... iter:  3
looping... iter:  4
looping... iter:  5
looping... iter:  6
looping... iter:  7
looping... iter:  8
looping... iter:  9
looping... iter:  10
looping... iter:  11
looping... iter:  12
Wow, target reached!!!
```