CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Security Testing

WS 2021/2022

Prof. Dr. Andreas Zeller
Leon Bettscheider
Marius Smytzek

Exercise 10 (10 Points)

Due: 23. January 2022

The lecture is based on The Fuzzing Book, an *interactive textbook that allows you to try out code right in your web browser*.
The Fuzzing Book code is additionally available as a Python pip package. To work on the exercises, please install the package locally:

```
pip3 install fuzzingbook
```

Submit your solutions as a Zip file on your status page in the CMS.

We will provide you a structure to submit your solutions where each task has a dedicated file. You can add new files and scripts if you want, but you may not delete any provided ones. You can verify whether your submission is valid by executing `verify.py`:

```
python3 verify.py
```

The output provides an overview if a required file, variable, or function is missing and if a function pattern was altered. If you do not follow this structure or change it, we cannot evaluate your submission. A non evaluable exercise will result in 0 points, so make sure to verify your work before submitting it. Note that the script does not reveal if your solutions are correct.

Exercise 10-1: Symbolic by Hand (10 Points)

In this exercise you will work on symbolic execution and testing to understand how these can be leveraged for fuzzing.

Over the course of this exercise you will work on the following function that is also implemented in **example.py**:

In [2]:
```
def foo(x: int, y: int, z: int) -> tuple[int, int]:

    if x > 2:
        a: int = 2
        b: int = 3
        if y <= 0 and z == 1:
            b: int = 2
    else:
        a: int = 0
        b: int = 5

    if y > 0:
        b: int = 7
    else:
        a: int = 3

    assert a + b != 5

    return a, b
```

a. Build a Tree (3 Points)

We start by building the execution tree of the path conditions in the function.

Consider the following function:

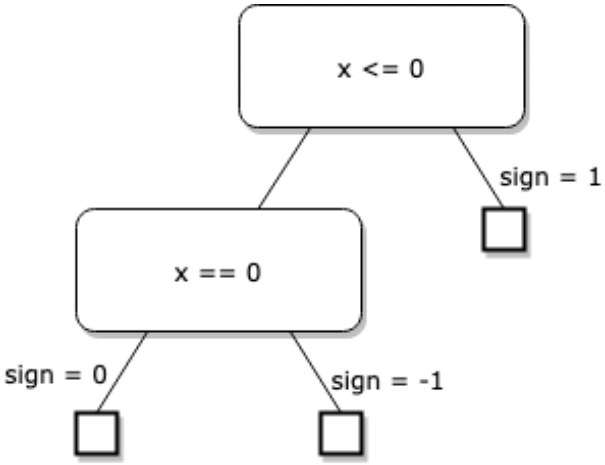
In [3]:
```
def f(x: int) -> int:
    if x <= 0:
```

```
        if x == 0:
            sign: int = 0
        else:
            sign: int = -1
    else:
        sign: int = 1

    return sign
```

An execution tree consists of nodes that represent the conditions along a path and edges that provide the action to variables that happen when the condition is `True` or `False`.

The execution tree of the above function looks like the following:



This is nice but we cannot work on it. Because of this, we provide a framework in **tree.py** that allows you to implement this tree structure.

We start by introducing the conditions.

All conditions, defined by the class `Condition`, provide a method `Condition.evaluate(self, values: dict)` that evaluates the conditions based on the variable assignments in `values`.

To define a variable, use the `Var` class together with the variable name.

In [4]:
```
var = Var('x')
var.evaluate({'x': 2})
```

Out[4]:
```
2
```

To define a constant, use the `Const` class together with the value of the constant. You can provide any value to this constructor.

In [5]:
```
const = Const(42)
const.evaluate({'x': 2})
```

Out[5]:
```
42
```

The framework provides the following comparisons `Lt` (`<`), `Le` (`<=`), `Gt` (`>`), `Ge` (`>=`), and `Eq` (`==`). All of them take two objects of the type `Condition` and compare the evaluations of these with the corresponding operator.

In [6]:
```
# x < 42
comp = Lt(var, const)
comp.evaluate({'x': 2})
```

Out[6]:
```
True
```

Besides comparisons, the framework allows you to apply the boolean operations `And` (`and`), `Or` (`or`), and `Not` (`not`). The two binary operators `And` and `Or` are used as follows:

In [7]:
```
# True and False
bop = And(Const(True), Const(False))
```

```
bop.evaluate({'x': 2})
```

Out[7]:
```
False
```

The unary operator `Not` is used like this:

In [8]:
```
# not False
bop = Not(Const(False))
bop.evaluate({'x': 2})
```

Out[8]:
```
True
```

Now we can translate the execution tree of the function `f()` from above.

We start by implementing the conditions:

In [9]:
```
cle = Le(Var('x'), Const(0))  # x <= 0
ceq = Eq(Var('x'), Const(0))  # x == 0
```

When tackling such a task it is always a good idea to follow a bottom-up approach. So we start with the leave nodes of the tree. Note that we do not need to translate the empty nodes explicitly.

To implement a node the framework provides a `Node` class. The constructor of this class takes the following arguments:

- `condition: Condition` : The condition for this node.
- `true_action: dict = None` : The changes in the variables, if the condition evaluates to `True` . If nothing changes, you can provide `None` , an empty dictionary, or you do not need to set this argument.
- `false_action: dict = None` : The changes in the variables, if the condition evaluates to `False` . If nothing changes, you can provide `None` , an empty dictionary, or you do not need to set this argument.
- `true_child: Node = None` : The child node to evaluate, if the condition evaluates to `True` . If there is no following condition, you can provide `None` or you do not need to set this argument.
- `false_child: Node = None` : The child node to evaluate, if the condition evaluates to `False` . If there is no following condition, you can provide `None` or you do not need to set this argument.

So we start with the conditions that do not have other conditions as children, which is in this case `x == 0` . If this evaluates to `True` , `sign` is set to `0` , and `-1` otherwise.

In [10]:
```
teq = Node(ceq, true_action={'sign': 0}, false_action={'sign': -1})
```

Then we will make the way up the tree. So we need to encode `x <= 0` . If this evaluates to `True` , we have no direct actions but `teq` as the corresponding child. If it evaluates to `False` , `sign` is set to `1` .

In [11]:
```
tle = Node(cle, false_action={'sign': 1}, true_child=teq)
```

Note that `Node` also provides a method `Node.evaluate(self, values: dict) -> dict` that returns a variable assignment when following the evaluation.

This allows us to evaluate our tree.

In [12]:
```
assert tle.evaluate({'x': 42})['sign'] == 1
assert tle.evaluate({'x': 0})['sign'] == 0
assert tle.evaluate({'x': -8})['sign'] == -1
```

Now is your turn to create the execution tree for the function `foo(x: int, y: int, z: int) -> tuple[int, int]` .

We would recommend to follow these steps:

- Create the tree by hand on paper.
- Translate all condition with the framework.
- Build the nodes of the tree as a bottom-up approach.
- The last node you created is the root of the tree.

Note that you can copy and reuse nodes and conditions but look out that you do not create loops.

Implement your tree in `exercise_1a.py` in the variable `execution_tree`. You are allowed to add new variables if you like.

You can evaluate your implementation by running `python3.9 test_1a.py`. Your points for this exercise will be given based on these tests.

b. Collect the Constraints (3 Points)

In this exercise you should find all path constraints of the path's in `foo(x: int, y: int, z: int) -> tuple[int, int]`. There are *6* unique paths in `foo()`. Find these paths with the help of your execution tree from ***Exercise 10-1 a.***. The nodes in the tree and the path you take to get to the node provide you the path constraint.

As an example, consider the execution tree of `f()` above. There are three paths, we will select the path where the first condition evaluate to `True` and the second to `False` and build the path constraint with the introduced framework:

In [19]:
```
example_path_constraint = And(Le(Var('x'), Const(0)), Not(Eq(Var('x'), Const(0))))
example_path_constraint
```

Out[19]:
```
((x) <= (0)) and (not ((x) == (0)))
```

Implement your solution in **exercise_1b.py**. Use the variables `path_constraint_1` to `path_constraint_6` for the path constraints. You are allowed to use variables you defined in **exercise_1a.py**.

You can see the constraints you build by running `python3.9 exercise_1b.py`.

*Hint: You only need `And` and `Not` of the Framework and the conditions defined in **Exercise 10-1 a.** to build the path constraints*.

c. Solve the Constraints (3 Points)

In this step you will solve your collected path constraints. Provide for each path constraint from **exercise_1b.py** a solution in **exercise_1c.py**, where `solution_1` correspond to your path constraint `path_constraint_1`, `solution_2` correspond to your path constraint `path_constraint_2`, and so on. A solution is a dictionary containing the variables `'x'`, `'y'`, and `'z'`. If a path constraint does not have a solution provide `None` as the solution.

A solution for the `example_path_constraint` from ***Exercise 10-1 b.*** would be:

In [26]:
```
example_solution = {'x': -1}
example_path_constraint.evaluate(example_solution)
```

Out[26]:
```
True
```

You can verify your solutions by running `python3.9 test_1c.py`.

Note that only unique path constraints and solutions count for this exercise.

d. Find the Bug (1 Point)

Find a solution, i.e. values for `x`, `y`, and `z` that trigger the `AssertionError` in `foo()`. If you solved the previous exercises, you only need to find the path constraint corresponding to the path that triggers this bug and use your solution from ***Exercise 10-1 c.***. Provide your solution in `exercise_1d.py` as the variables `x`, `y`, and `z`.

You can run `python3.9 test_1d.py` to verify that your solution triggers the error.