CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Security Testing

WS 2023/2024

Prof. Dr. Andreas Zeller
Leon Bettscheider
José Antonio Zamudio Amaya

Exercise 3 (10 Points)

Due: 26. November 2023 23:59

> 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 3-1: To not see the Forest for the Trees (4 Points)

In this exercise you will work on derivation trees and need to develop an algorithm that could help to generate new trees. Even though, your algorithm should not be limited to a single grammar, we only consider derivation trees of the `EXPR_GRAMMAR` from Fuzzing with Grammars.

a. Hide and Seek (2 Points)

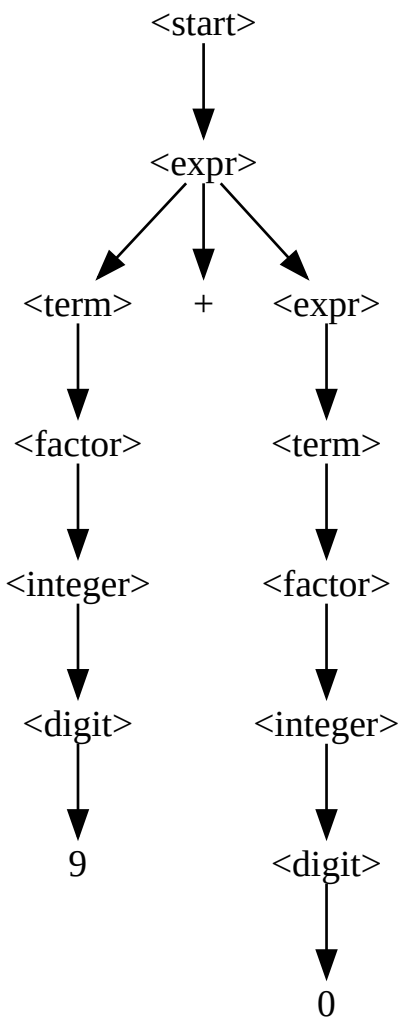
The first step of this algorithm is to search all existing subtrees that are derived from the a given nonterminal in a provided tree. For this you should implement the function `find_subtrees(tree, symbol)` in **exercise_1a.py**. To find the subtrees you need to traverse the given derivation tree `tree` and collect all subtrees from it derived from the nonterminal `symbol`. Collect all found subtrees in a list and return this list.

Consider the following derivation tree of `9 + 0`:

In [3]:
```
tree = ('<start>', [
    ('<expr>', [
        ('<term>', [
```

```
            ('<factor>', [
                ('<integer>', [
                    ('<digit>', [('9', None)])
                ])
            ])
        ]),
        (' + ', []),
        ('<expr>', [
            ('<term>', [
                ('<factor>', [
                    ('<integer>', [
                        ('<digit>', [('0', None)])
                    ])
                ])
            ])
        ])
    ])
])
GrammarFuzzer.display_tree(tree)
```

Out[3]:



Consider we want to find all `<integer>` subtrees, the result of the function `find_subtrees(tree, '<integer>')` should then be the `<integer>` subtrees as a list:

```
[('<integer>', [('<digit>', [('9', None)])]), ('<integer>', [('<digit>', [('0', None)])])]
```

You are free to implement the function however you like. It does not matter if you implement it as a BFS, a DFS, recursive, or iterative. You are also allowed to implement helper functions. As long as you return a list containing all relevant subtrees, you are good to go.

The file **trees.py** contains two derivation trees you can leverage for evaluating your function.

b. Search and Replace (2 Points)

For this part you need to implement the function `replace_random_subtree(tree, symbol, subtrees)` in **exercise_1b.py**. The function should replace a random subtree in `tree` that is derived from the nonterminal `symbol` with a random subtree in the list of possible subtrees `subtrees`. You can leverage your implementation of `find_subtrees(tree, symbol)` to find possible subtrees in `tree` to replace. You can consider that all subtrees in `subtrees` are derived from the nonterminal `symbol` and your function does not need to validate this.

Consider the derivation tree for `9 + 0` from **exercise 3-1 a.** as the argument `tree`, `'<digit>'` as `symbol` and

```
[('<digit>', [('1', None)]), ('<digit>', [('2', None)])]
```

as the possible `subtrees`. A possible return of the function `replace_random_subtree(tree, symbol, subtrees)` could be `9 + 2` or as a derivation tree:

```
('<start>', [
    ('<expr>', [
        ('<term>', [
            ('<factor>', [
                ('<integer>', [
                    ('<digit>', [('9', None)])
                ])
            ])
        ]),
        (' + ', []),
        ('<expr>', [
            ('<term>', [
                ('<factor>', [
                    ('<integer>', [
                        ('<digit>', [('2', None)])
                    ])
                ])
            ])
        ])
    ])
])
```

In this case the function had selected `('<digit>', [('0', None)])` as the subtree to replace and then replaced it with `('<digit>', [('2', None)])` from `subtrees`.

Like in **exercise 3-1 a.**, you can approach the implementation however you like and you are allowed to introduce helper functions.

c. Fuzzing with Trees (0 Points)

We have implemented a fuzzer that leverages your `find_subtrees()` and `replace_random_subtree()` functions, to extend a set of seed trees and generate new trees by replacing subtrees. You can find the fuzzer in **exercise_1c.py**. You can try it by running `python3 exercise_1c.py` and see your algorithm in action.

Exercise 3-2: Cover (.*?) (3 Points)

In this exercise you will take a look at a subset of Python's regular expressions. If you are unfamiliar with regular expressions in Python you can take a look at this tutorial. You can find a grammar for this subset in **exercise_2.py**.

In [22]:
```
RE_GRAMMAR = {
    '<start>': ['<alternative>', '^<alternative>', '<alternative>$',
                '^<alternative>$'],
    '<alternative>': ['<concat>', '<concat>|<alternative>'],
    '<concat>': ['', '<concat><regex>'],
    '<regex>': ['<symbol>', '<symbol>*', '<symbol>+', '<symbol>?',
                '<symbol>{<range>}'],
    '<symbol>': ['.', '<char>', '(<alternative>)'],
    '<char>': ['a', 'b', 'c'],
    '<range>': ['<num>', ',<num>'],
    '<num>': ['1', '2'],
}
```

You will dig into grammar coverage and fuzz with it. As part of this exercise you will develop a fuzzing experiment that shows how grammar coverage and systematically expanding a grammar could improve the code coverage based on Python's `re` module.

a. Pump it up (1 Point)

Expand the grammar from **exercise_2.py** such that it can provide more context specific coverage as explained in Coverage in Context. You can approach this manually or programmatically. Provide your expanded grammar as `RE_GRAMMAR_EXPANDED` in **exercise_2a.py**

b. Experiment with it (2 Points)

Setup an experiment to evaluate your expanded grammar from **exercise 3-2 a.** and implement it in **exercise_2b.py**. In this experiment you want to compare the `GrammarFuzzer` and `GrammarCoverageFuzzer` with the `RE_GRAMMAR` grammar against the `GrammarCoverageFuzzer` with your `RE_GRAMMAR_EXPANDED` grammar. We provide you a `get_coverage(fuzzer)` function in **re_coverage.py** that takes a fuzzer and computes the line coverage it achieves on the `re.compile()` function. You are not allowed to modify the `get_coverage()` function.

For each of the fuzzer configurations do the following:

- Set up the fuzzer with its corresponding grammar.
- Calculate the code coverage it achieves with the `get_coverage()` function.
- Repeat these steps a total of **25** times and calculate the average of the coverage values.

At the end print the averages for all fuzzer configurations as shown in **exercise_2b.py**.

Exercise 3-3: Questions (3 Points)

Provide the BEST answers to the following questions in `exercise_3.py` by assigning to each variable `Q1, ..., Q6` the values `1` to `6`.

For instance, if you think the first answer is the BEST answer to Q1, set `Q1=1` in `exercise_3.py`.

There is only **one BEST answer** to each question.

Question 1: What is the primary purpose of using a `tree-based algorithm` in grammar fuzzing compared to a `string-based algorithm`? (0.5p)

- 1. To decrease the complexity of the algorithm, making it faster than string-based algorithms.

- 2. To allow for a more efficient and controlled production of fuzz inputs.
- 3. To produce longer and more rich strings inputs for testing.
- 4. To simplify the selection mechanism of expansion rules.

Question 2: In the context of grammar fuzzing, what is the role of `derivation trees`? (0.5p)

- 1. To record the structure and production history of the produced string.
- 2. To limit the size of the input strings.
- 3. To replace the need for a grammar in generating test cases.
- 4. To visualize the grammar rules.

Question 3: What is the role of the `expand_node_max_cost()` method in the GrammarFuzzer class? (0.5p)

- 1. It expands a node at minimum cost to avoid infinite expansions.
- 2. It expands a node randomly without considering costs.
- 3. It expands a node at maximum cost to get as many nodes as possible.
- 4. It sets the maximum number of nonterminals for the fuzzer.

Question 4: In the GrammarFuzzer's `expand_tree()` method, which of the following best describes the sequence and purpose of the three expansion rules? (0.5p)

- 1. Max cost expansion for minimal nodes, Random expansion for controlled growth, Min cost expansion to finalize the tree.
- 2. Random expansion for initial growth, Max cost expansion for adding complexity, Min cost expansion to prevent infinite loops.
- 3. Max cost expansion for initial growth, Random expansion for adding complexity, Min cost expansion to finalize the tree.
- 4. Sequential expansion for initial growth, Random expansion for node balancing, Max cost expansion to close the tree.

Question 5: Why is expanding the grammar and fuzzing it with grammar coverage in mind beneficial for improving code coverage? (0.5p)

- 1. Expanding grammar allows for more complex test cases, directly boosting code coverage.
- 2. Duplicating rules in specific contexts provides a handle to cover expansions in different contexts, leading to increased code coverage.
- 3. Grammar coverage focuses on reaching individual expansions, which is sufficient for comprehensive code coverage.
- 4. Code coverage and grammar coverage are unrelated; expanding grammar has no impact on code coverage.

Question 6: How does the `GrammarFuzzer` address the inefficiencies of `simple_grammar_fuzzer()`? (0.5p)

- 1. By producing strings of a predetermined length.
- 2. By controlling expansions with min and max nonterminals.
- 3. By eliminating the need for a grammar in generating fuzz inputs.
- 4. All of the above.