Security Testing

WS 2023/2024

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

Exercise 4 (10 Points)

Due: 3. December 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 4-1: Of Snakes and Dwarfs (5 Points)

a. Snake (2 Points)

Given the following grammar:

In [3]:
```python
SNAKE_GRAMMAR = {
    '<start>': ['<snake>', ''],
    '<snake>': ['<digit>', '<snake><digit>', ''],
    '<digit>': ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
}
```

Add probabilities to this grammar such that the empty string `''` will never be produced when feeding this grammar to a probabilistic grammar fuzzer and the probability of adding a new digit to the snake is *6* times higher than ending the snake with a digit. Provide your annotated grammar in **exercise_1a.py**.

b. Heigh-Ho (2 Points)

For this exercise you will work on learning probabilities from samples for an existing grammar. Consider the following grammar for regular expressions, as in **exercise 3-2** this is only a subset of the language.

In [5]:
```
RE_GRAMMAR = {
    '<start>': ['<re>'],
    '<re>': ['<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>,'],
    '<num>': ['1', '2'],
}
```

We had a beautiful grammar with probabilities but unfortunately we lost it. Luckily we were able to derive examples from this grammar, you can find these *2500* examples in **examples.py**. Can you help us to reconstruct the lost grammar?

To accomplish this please implement the function `learn_probabilities()` in **exercise_1b.py**. The function should mine the probabilities for the `RE_GRAMMAR` from the *2500* examples and should return the mined probabilistic grammar.

c. Who introduced 16 decimals!? (1 Points)

The original grammar from where `examples` was generated was as follows:

In [10]:
```
RE_GRAMMAR_PROB = {
    '<start>': ['<re>'],
    '<re>': [('<alternative>', opts(prob=0.6)),
            ('^<alternative>', opts(prob=0.1)),
                ('<alternative>$', opts(prob=0.1)),
                ('^<alternative>$', opts(prob=0.2))],
    '<alternative>': [('<concat>', opts(prob=0.4)),
                    ('<concat>|<alternative>', opts(prob=0.6))],
    '<concat>': [('', opts(prob=0.1)), ('<concat><regex>',
                                        opts(prob=0.9))],
    '<regex>': [('<symbol>', opts(prob=0.6)), ('<symbol>*',
                                        opts(prob=0.125)),
                ('<symbol>+', opts(prob=0.125)), ('<symbol>?',
                                        opts(prob=0.1)),
                ('<symbol>{<range>}', opts(prob=0.05))],
    '<symbol>': [('<char>', opts(prob=0.64)), ('(<alternative>)',
                opts(prob=0.35)), ('.', opts(prob=0.01))],
    '<char>': ['a', 'b', 'c'],
    '<range>': [('<num>', opts(prob=0.8)), ',<num>', '<num>,'],
    '<num>': ['1', '2'],
}
```

However, you can see that the probabilities you have obtained may vary from the original probabilities. They should be pretty close, but not exactly equal. This araises the following question: Why is that?

Provide the BEST answers to the following question in `exercise_1c.py` by assigning the variable `Q1` the values `1` to `4`.

For instance, if you think the first answer is the BEST answer to Q1, set `Q1=1` in `exercise_1c.py`.

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

Question 1: Why do some probabilities vary from the original probabilities? (1p)

- 1. It is a consequence of reducing cost during the closing phase of the probabilistic grammar fuzzer.
- 2. We need many more sample inputs to learn the probability more precisely.
- 3. Because of the replication in the `ProbabilisticGrammarMiner` probability distribution.
- 4. All of the above.

Exercise 4-2: How to get Rich? (5 Points)

In this exercise you will work with International Bank Account Numbers or short IBANs. An IBAN consists of a country code (two uppercase letters), two check digits, and a country specific Basic Bank Account Number (BBAN). You will not concentrate on the entire set of possible IBANs, but we limited BBANs to only consist of digits instead of an alphanumerical string. We will provide you a grammar for IBANs that you should extend with generators to produce valid IBANs. Provide all your solutions for this exercise in **exercise_2.py**.

a. Countries and Lengths (2 Points)

Provide a generator or multiple generators in the grammar to produce only valid country codes and BBANs with the country specific length. **exercise_2.py** contains a list `iban_cc_len` of valid country codes for all countries using IBAN and the entire length of the IBAN. Note that this length contains the country code and the two check digits. We would recommend to randomly select one element of `iban_cc_len` at the beginning of a generator. You do not have to set the random seed for this exercise. Select smart positions to add your generator or generators.

b. Check (3 Points)

Add a generator or multiple generators to the grammar to produce valid check digits. The algorithm for calculating the check digits for an IBAN already containing a country code and a BBAN is described as following (source):

1. Check that the total IBAN length is correct as per the country. If not, the IBAN is invalid. In this case you should raise a `ValueError()` exception. If this happens during your generation you should overwork your solution for **exercise 4-2 a.** because the length should be correct at this point.
2. Replace the two check digits by 00 (e.g., GB00 for the UK).
3. Move the four initial characters to the end of the string.
4. Replace the letters in the string with digits, expanding the string as necessary, such that A or a = 10, B or b = 11, and Z or z = 35. Each alphabetic character is therefore replaced by 2 digits
5. Convert the string to an integer (i.e. ignore leading zeroes).
6. Calculate mod-97 of the new number, which results in the remainder.
7. Subtract the remainder from 98 and use the result for the two check digits. If the result is a single-digit number, pad it with a leading 0 to make a two-digit number.

Select suited positions to add your generator or generators

Final thoughts

What you have achieved in this exercise could be easily accomplished programmatically, i.e. without the usage of a grammar but instead writing code to generate an IBAN. But using a grammar provides benefits over such an approach. The main advantages for this are:

- Maintainability: A grammar is easier to maintain, adopt, or extend to new problems or changes in the specification.
- Readability: A grammar is human readable, i.e., a human being can faster understand a grammar than a program (if it knows the syntax).
- Grammar Fuzzing: We can apply other grammar fuzzing techniques to the grammar, for instance, probabilities or other generators, and need only little modifications to adopt it.