Security Testing

WS 2021/2022

Prof. Dr. Andreas Zeller
Leon Bettscheider
Marius Smytzek

Exercise 6 (10 Points)

Due: 12. December 2021

The lecture is based on The Fuzzing Book (https://fuzzingbook.org/beta), 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 (https://cms.cispa.saarland/fuzzing2122/students/view).

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 6-1: I Me Mine (5 Points)

In this exercise you will use **input grammar mining** to mine and generalize a grammar.

To familiarize yourself with the concept of mining a grammar, please read the chapter on Grammar Mining (https://www.fuzzingbook.org/html/GrammarMiner.html).

a. Mine it (1 Points)

In this exercise, you should mine the grammar of a custom parser that parses *student records* given in a subset of JSON. Please use the function `recover_grammar` from the chapter mentioned above to recover the grammar

The grammar mining should be carried out on the following *student records* (stored in `inputs.py`):

```
INPS = [
    '{"name": "Tron", "subject": "cybersecurity", "termsofstudying": "780", "matnr":
"7009842"}',
    '{"name": "Tina", "subject": "cybersecurity", "termsofstudying": "501", "matnr":
"7009844"}',
    '{"name": "Leon", "subject": "computer science", "termsofstudying": "640", "matnr":
"2578921"}',
    '{"name": "Martin", "subject": "cybersecurity", "termsofstudying": "504", "matnr":
"7009862"}',
    '{"name": "Josephine", "subject": "cybersecurity", "termsofstudying": "508", "matnr":
"7005847"}',
    '{"name": "Don", "subject": "music", "termsofstudying": "100", "matnr": "2578542"}',
    '{"name": "Andreas", "subject": "computer science", "termsofstudying": "10000",
"matnr": "2574572"}',
    '{"name": "Paul", "subject": "music", "termsofstudying": "456", "matnr": "2578533"}',
    '{"name": "Marius", "subject": "computer science", "termsofstudying": "787", "matnr":
"2578922"}',
    '{"name": "Nena", "subject": "music", "termsofstudying": "786", "matnr": "2578549"}',
]
```

The following parser is stored in `parser.py` and should be used to mine the grammar from.

```python
def parse(s):
    try:
        j = json.loads(s)
        name = j['name']
        subject = j['subject']
        termsofstudying = j['termsofstudying']
        matnr = j['matnr']
        assert isinstance(name, str) and isinstance(subject, str) \
             and isinstance(termsofstudying, str) and isinstance(matnr, str)
        return name, subject, termsofstudying, matnr
    except Exception:
        return None
```

Implement the function

```python
def mine_student_grammar() -> dict
```

in file `exercise_1a.py` which returns the grammar that is mined from the function `parse` using the inputs `INPS`. The implementation should only take one line. You should import `parse` and `INPS` from their respective file.

b. Generalize the Grammar (4 Points)

While the grammar generated by the grammar miner in the previous exercise is able to *recombine* input fragments, it does not *generalize* input fragments. For instance, it cannot come up with *new* matriculation numbers (*matnr*), but only reuses observed ones.

In this exercise you will implement a function

```python
def generalize(g: dict, cnt_inputs: int) -> dict
```

in file `exercise_1b.py` which iterates over all keys and rules in the grammar `g` and performs the generalization according to the following rules:

```
- If the key has fewer than `len(cnt_inputs)/2` distinct rules:
    - Do not change its rules.
- Else:
    - If the key has at least one rule that contains a non-terminal:
        - Do not change its rules.
    - Else:
        - If all strings produced by the key are (sequences of) digits:
            - Replace its rules with a rule that produces only sequences of digits (0-9)
(len>=1).
        - Else:
            - If all strings produced by the key are (sequences of) letters (a-zA-Z):
                - Replace its rules with a rule that produces only sequences of letters (a-
zA-Z) (len>=1).
```

As an example, the resulting grammar should be able to produce inputs such as:

```
{"name": "spWEk", "subject": "cybersecurity", "termsofstudying": "769", "matnr": "8"}
{"name": "ROI", "subject": "music", "termsofstudying": "7", "matnr": "92118"}
```

Note that the argument `cnt_inputs` should be the number of inputs, i.e. `len(INPS)`. You are allowed to implement helper functions.

Exercise 6-2: Recap (5 Points)

In this exercise we will recap different aspects of grammar-based blackbox fuzzing that we have seen so far.

Provide the BEST answers to the following questions in `exercise_2.py` by assigning to each variable `Q1, ..., Q10` the values `1` to `4`.

For instance, if you think the first answer is the BEST answer to Q1, set `Q1=1` in `exercise_2.py`.

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

Questions

Q1: What information is given by a derivation tree?

1. It is another representation of a grammar.
2. The structure of all possible inputs according to a grammar.
3. The structure of a program when parsing an input.
4. The structure of an input according to a grammar.

Q2: How are probabilities used in Probabilistic Grammar Fuzzing?

1. They control how deep the expansion tree grows.
2. They control how many expansions are chosen.
3. They control how many tokens appear in each expansion.
4. They control how often each expansion is chosen.

Q3: Which of the following would be a good probability distribution if we want to generate on average as many as, bs and cs?

```
<start> ::= <letters>
<letters> ::= <letter> | <letter><letters>
<letter> ::= <A><A> | <B><B> | <C>
<A> ::= a
<B> ::= b
<C> ::= c
```

1. `<letter> ::= <A><A> (0.25) | <B><B> (0.25) | <C> (0.5)`
2. `<letter> ::= <A><A> (0.3333) | <B><B> (0.3333) | <C> (0.3333)`
3. `<letter> ::= <A><A> (0.5) | <B><B> (0.5) | <C> (0.5)`
4. `<letter> ::= <A><A> (0.15) | <B><B> (0.15) | <C> (0.7)`

Q4: Which of the following languages CANNOT be expressed with a context-free grammar?

1. The language of correctly closed HTML tags (featuring a finite set of tag names).
2. The language of correctly closed XML tags (featuring an infinite set of tag names)
3. The language of balanced parentheses (for example (()(()())()) is a balanced parentheses expression)
4. The language containing all the HTML pages hosted at www.fuzzingbook.org.

Q5: Why is it sometimes useful to duplicate non-terminals while using the Grammar Coverage criteria?

1. To cover the use of the non-terminals in different contexts.
2. To keep generating inputs even when all the original grammar is covered.
3. To have a higher proportion of these duplicated non-terminals.
4. To avoid problems with recursive expansion rules.

Q6: What is the criteria defined in the lecture for Grammar Coverage?

1. Generate inputs to cover all possible non-terminals.
2. Generate inputs to cover all possible terminals.
3. Generate inputs to cover all possible expansions.
4. Generate inputs to cover all possible symbols (terminals and non-terminals).

Q7: Which of the following techniques is presented in the lecture to learn probabilities for grammars automatically?

1. Learn probabilities from branches taken in the target program.
2. Learn probabilities from another grammar for a similar input format.
3. Learn probabilities by removing unexplored parts of the grammar.
4. Learn probabilities from input samples.

Q8: How can one best focus on testing uncommon features of a probabilistic grammar?

1. Invert the distribution of probabilities.
2. Increase all probabilities.
3. Swap probabilities of two or more production rules.
4. Distribute probabilities equally between alternative expansions.

Q9: Which coverage criteria guarantees that all bugs in a program will be found?

1. Branch coverage.
2. Statement coverage.
3. There exist no such criteria.
4. Either branch or statement coverage.

Q10: Given an infinite amount of time, which technique is most likely to perform best?

1. Probabilistic grammar-based fuzzing.
2. Random Fuzzing.
3. Grammar-based fuzzing with a set of valid seed inputs.
4. Grammar-based fuzzing with generators.