CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

Security Testing
WS 2023/2024

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

Project 2 - Learning constraints (Version 2)

Due: 18. February 2024 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
```

Motivation

Welcome to the world of high-stakes digital heist! You have recently joined a group of elite hackers, which have stumbled upon a golden opportunity at "DefinitelyNotCISPA GmbH". However, they want to test you. They are not sure about your hacking abilities, so what is your mission? Infiltrate the company's internal network through an unlikely entry point - their printers?!

Your initial plan is simple yet daring: connect a fake printer (in reality, your trusty computer) to the network and find your way into the system. But as every heist story goes, there was a twist - a vigilant gateway standing guard over the printer connections, blocking our every move.

Just when our hopes were dwindling, luck favored the bold. Amidst our relentless attempts, we noticed something peculiar - sometimes, just sometimes, our fake printer managed to connect. This inconsistency was the chink in the armor we were looking for. We gathered samples of these successful and unsuccessful attempts, our key to unraveling the gateway's secrets. Enter the master plan: 'The Perfect Fuzzer'. More than just a tool, it would be a masterpiece of digital craftsmanship. This fuzzer would learn from our samples, decode the gateway's constraints, and then, in a stroke of genius, create a grammar so precise, so flawless, that it would always be able to connect to this shady system.

This will be your test of skill, wit, and cunning. If you succeed, you'll have full access to the network - a hacker's paradise full of secrets, data, and, of course, unlimited printer ink. In the world of hacking, they say, "A printer today, the world tomorrow!" Let's show them how right they are.

Project Description

In this project, you will build an **automated grammar-based constraint learning system**. Your task is to implement a program that *learns constraints* from a provided set of positive and negative sample inputs. Once learned, the constraints can be used to **check** whether a given input satisfies them and to **generate** inputs that satisfy them (or: do not satisfy them if desired).

Learning such input constraints has applications in:

- Program testing: Finding the conditions under which an input is accepted by an **input validation** component, which allows to fuzz deeper.
- Program repair: Finding the conditions under which an input triggers a bug.

Core idea

We provide you with the following information:

- a set of unary and binary operations
- a context-free grammar
- a set of passing (=positive) samples (which satifsy the unknown constraints)
- a set of failing (=negative) samples (which do not satisfy the unknown constraints)
- a blackbox oracle, which decides whether a given input satisfies the unknown constraints
- a set of helper methods which you can use in your implementation

Example

Consider the following grammar:

In [ ]:
```
grammar = {"<start>": ["<student>"],
           "<student>": ["<firstname>,<lastname>,<age>"],
           "<firstname>": ["<string_lowercase>"],
           "<lastname>": ["<string_uppercase>"],
           "<string_lowercase>": ["<char_lowercase>",
              "<char_lowercase><string_lowercase>"],
           "<string_uppercase>": ["<char_uppercase>",
              "<char_uppercase><string_uppercase>"],
           "<char_lowercase>": ["a", "b", "...", "z"],
           "<char_uppercase>": ["A", "B", "...", "Z"],
           "<age>": ["1", "2", "3", "...", "99"]
    }
```

And the following sets of passing and failing inputs:

In [ ]:
```
passing_inputs = ["andreas,SMITH,7",
                  "leon,DOE,4",
                  "jose,MILLER,4",
                  "harry,POTTER,5"]

failing_inputs = ["john,JOHNES,9",
                  "frank,MILLER,3",
```

```
                    "mike,MONROE,6",
                    "jules,DEAN,4"]
```

And the following set of candidate unary or binary constraint patterns. The `{}` symbols are placeholders and will later be instantiated with specific non-terminal symbols.

In [ ]:
```
constraint_patterns = ["str({}) == str({})",
                       "len(str({})) == int({})",
                       "int({}) == 0"]
```

And an unknown oracle which decides whether a given input (which can be parsed by the grammar) is passing or failing. For instance `oracle("andreas,SMITH,7") == True` and `oracle("john,JOHNES,9") == False`.

The passing and failing puts were already classified using the `oracle`. However, one could generate more inputs and classify them using the provided `oracle`.

Task

The task is now to instantiate these constraint patterns with specific non-terminal symbols, and to check which instantiated constraints (=abstract constraints) hold for all of positive inputs, and none of the negative inputs. The result of the task is a set of abstract constraints that hold.

Next, we will run you through the individual steps using a concrete example. While we're using concrete values here, your approach must be general. For example, you may not hard-code the non-terminal symbols as we do here.

Step 1: Get the set of non-terminal symbols that occur in the derivation trees of every passing and failing inputs.

In [ ]:
```
non_terminals = set(["<start>", "<student>",
                     "<firstname>", "<lastname>", ...])
```

Step 2: Instantiate each constraint pattern with each combination of non-terminal symbols.

In [ ]:
```
abstract_constraints = [
# This set must be generated automatically by your implementation.
    "str(<start>) == str(<start>)",
    "str(<start>) == str(<student>)",
    ...,
    "str(<student>) == str(<start>)",
    "str(<student>) == str(<student>)",
    ...,

    "len(str(<start>)) == int(<start>)",
    "len(str(<start>)) == int(<student>)",
    ...,
    "len(str(<student>)) == int(<start>)",
    "len(str(<student>)) == int(<student>)",
    ...,

    "int(<start>) == 0",
    "int(<student>) == 0",
```

```
    "int(<firstname>) == 0",
    "int(<lastname>) == 0",
    ...
    ]
```

Step 3: For each of the abstract constraints, check which of them hold for all passing inputs.

We can do this as follows:

- We iterate over all abstract constraints. In this example, let's fix one iteration and assume the current abstract constraint is `len(str(<lastname>)) == int(<age>)`.
- Find all nonterminal symbols that occur in the current abstract constraint. In this case: `<lastname>` and `<age>`.
- Parse every passing input into a derivation tree, and store all subtrees rooted at `<lastname>` or `<age>`. In this example, let's fix one input again: `harry,POTTER,5`.
- Instantiate the abstract constraints one more time, this time replacing nonterminals with the terminals of each matching subtree. In this example, we have one subtree rooted in `<lastname>` with the terminals `POTTER` and one subtree rooted in `<age>` with the terminals `5`. Hence, the result of this in operation, which we call a `concrete constraint` is `len(str("POTTER")) == int("5")`. Note that there could be more than one `concrete constraints` at this point because, depending on the grammar, there could be multiple subtrees rooted in `<lastname>` or `<age>`. We say an abstract constraint, such as `len(str(<lastname>)) == int(<age>)`, holds for a derivation tree if it holds for at least one combination of matching subtrees rooted at the respective non-terminal symbols.
- Evaluate each `concrete constraint` using the builtin `eval` function. In this case:

In [ ]:
```
eval('len(str("POTTER")) == int("5")') == False
```

- As soon as an abstract constraint (here: `len(str(<lastname>)) == int(<age>)`) does not hold (as indicated by the return value of eval) for any single input, we discard it, and continue with the next abstract constraint.

At the end of this procedure, we would find out that the abstract constraint `len(str(<firstname>)) == int(<age>)` holds for every passing inputs.

Step 4: For each of the abstract constraints, check which of them hold on the failing inputs.

Analogous to Step 3, except that the result is now the set of `abstract constraints` that hold for all failing inputs.

Step 5: Compute the set difference of the results of Step 3 and Step 4.

Result: The set of abstract constraints that hold for all passing inputs, and none of the failing inputs.

In this case, the desired constraint is: `len(str(<firstname>)) == int(<age>)`

Note that in general, the desired solution is a set of constraints, hence a conjunction of constraints can also be a solution (but not a disjunction).

What can we do with the learned constraint?

- The constraint can be used to **check** whether a given input would be accepted by the oracle. This can also be used for *filtering* failing inputs. One way to implement this is to parse the given input to a derivation tree, and to generate `concrete constraints` from the `abstract constraint` that we want to check. This can be done in a similar way as shown above in the steps for *learning constraints*. Next, the `concrete constraints` can be checked using `eval`.
- The constraint can be used to **generate** more passing or failing inputs on demand. One way to implement this is to generate new inputs with a grammar-based fuzzer until one of them is accepted by **check** as outlined above.

Your task in this project is to write a program that determines the correct set of `abstract constraints`.

Implementation

Your program should be implemented in `implementation.py`. The signatures of the functions in this file must not be the changed.

In [ ]:
```
def instantiate_with_nonterminals(constraint_pattern: str,
                                  nonterminals: list[str]) -> set[str]:
    pass

def instantiate_with_subtrees(abstract_constraint: str,
                              nts_to_subtrees: dict) -> set[str]:
    pass

def learn(constraint_patterns: list[str],
          derivation_trees: list) -> set[str]:
    pass

def check(abstract_constraints: set[str],
          derivation_tree) -> bool:
    pass

def generate(abstract_constraints: set[str],
             grammar: dict, produce_valid_sample: True) -> str:
    pass
```

Implementation details are up to you. We only require you to meet the requirements, function signatures and interfaces stated on this sheet:

instantiate_with_nonterminals

The `constraint_pattern` argument of this function are the individual elements of the `constraint_pattern` list of main.py. This function should instantiate the

placerholders `{}` in `constraint_pattern` with all combinations of nonterminals.

Example 1:

- `instantiate_with_nonterminals("int({}) > 0", ["<expr>", "<factor>"]) == set(["int(<expr>) > 0", "int(<factor>) > 0"])`

Example 2:

- `instantiate_with_nonterminals("{} == {}", ["<expr>", "<factor>"]) == set(["<expr> == <expr>", "<expr> == <factor>", "<factor> == <factor>", "<factor> == <expr>"])`

---

instantiate_with_subtrees

The `abstract_constraint` argument of this function are the individual outputs of the `instantiate_with_nonterminals` function, such as `<expr> > 0`. That is, there are no `{}` placeholders in `abstract_constraint`, but nonterminals instead.

`nts_to_subtrees` is a dictionary that maps nonterminal symbols to a list of subtrees rooted at that nonterminal symbol.

This function should instantiate the nonterminals found in `abstract_constraint` with all combinations of the terminals of the possible matching subtrees in `nts_to_subtrees`. We call the results of this function `concrete constraints`.

Example:

- `instantiate_with_subtrees("int(<A>) > 0", {"<A>": [subtree1, subtree2], "<B>": [subtree3, subtree4]}) == set([f"int({tree_to_string(subtree1)}) > 0", f"int({tree_to_string(subtree2)}) > 0"])`

---

learn

Given a list of `constraint_patterns` with placeholders, such as `["len(str({})) == int({})", "int({}) == 0"]`, and a list of `derivation_trees` (e.g., returned by the EarleyParser), `learn` should find a set of `abstract_constraints` instantiated with `nonterminals` (as returned by `instantiate_with_nonterminals`), which hold for all trees in `derivation_trees`.

The candidate nonterminal symbols are those nonterminal symbols which occur in every derivation tree.

Note that a tree might contain multiple occurences of the same nonterminal. Hence, we consider an abstract constraint such as `int(<expr>) > 0` to be satisfied if it holds for **at least one subtree rooted in `<expr>`** (existential quantifier).

---

check

Given a set of `abstract_constraints` and a single `derivation_tree`, this function returns True iff all abstract constraints hold on the derivation tree, and False otherwise. See the remarks on `learn` above for our definition of when a constraint is considered satisfied.

Note:

- Use the `eval` function to evaluate a `concrete constraint` (as returned by `instantiate_with_subtrees`).

Example:

- `check(set(["int(<some_number>) > 0"]), ("<some_number>", [("1", [])])) == True`

---

generate

Given a set of `abstract_constraints`, a `grammar` and the bool `produce_valid_sample`, `generate` returns an input (which can be parsed by `grammar`) that satisfies all abstract constraints if `produce_valid_sample==True`, or which does not satisfy all abstract constraints if `produce_valid_sample==False`.

In this project, the constraint are always simple enough such that you can generate satisfying inputs by generating random inputs with a grammar fuzzer and filtering satisfying inputs using the `check` function.

Example:

- `generate(set("int(<some_number>) > 0"), grammar) could return for instance "42" (depends on the grammar).`

We provide unit tests for each of these five functions in tests.py. You can run these tests via main.py. You can also run the entire constraint learning approach via main.py. See the `Framework` section for instructions.

Helper functions

We provide the following helper functions in `helpers.py`. Feel free to use them in your implementation.

- is_nt

- get_all_subtrees
- tree_to_string
- read_inputs

These FuzzingBook functions are also helpful:

- nonterminals (`from fuzzingbook.Grammars import nonterminals`)

Constraint pattern catalogue

Your implementation should use the following constraint patterns, which can also be found in `main.constraint_patterns`.

Unary operations

```
- int({}) == 0
- int({}) == 1
- int({}) != 0
- int({}) != 1
- int({}) >= 0
- int({}) >= 1
- len({}) == 0
- len({}) == 1
```

Binary operations

```
- str({}) == str({})
- str({}) in str({})
- str({}) not in str({})
- len(str({})) == int({})
- len(str({})) == len(str({}))
- len(str({})) < len(str({}))
- len(str({})) ^ len(str({})) == 0
- len(str({})) ^ len(str({})) == 1
- len(str({})) ^ int({}) == 0
- len(str({})) ^ int({}) == 1
- bool(int({})) ^ bool(int({}))
```

Goal

We provide an implementation of the function `main.learn_and_refine` which returns a set of constraints learned from its arguments: an oracle, a set of positive and negative samples, and a grammar. This implementation is based on the five functions you should implement: `check, learn, generate, instantiate_with_nonterminals, instantiate_with_subtrees`. If you implemented these functions correctly, `main.learn_and_refine` will return an accurate set of learned abstract constraints over the predefined set of operations and nonterminals found in the samples.

We will grade your submission by running `python3 main.py --learn-oracle-{i}` where `{i}` is the id of each oracle. This invokes the `main.learn_and_refine` function with the required arguments. Please ensure that your code can be executed this way and outputs an accurate set of constraints for each oracle.

The abstract constraints to be learned are one or more unary or binary operations, instantiated with nonterminal symbols, which must hold on **all** positive samples and **none** of the negative samples. If multiple `abstract constraints` hold, this means that they hold `in conjunction`.

You do not have to implement operations nested inside each other.

Note

- The constraints you will learn are simple. Hence, in order to `generate` inputs that satisfy the constraint, you can *always* randomly generate inputs from the grammar, and *filter* those that satisfy the constraint. To filter, you can use the `check` function.
- Since we provide you with the source code of the *blackbox* oracles, you can read the constraint that you should learn. However, do not make use of this information, and consider the oracles as a `blackbox`. In the evaluation, we will use different oracles, so overfitting to the provided ones will get you no points.
- Similarly, you should not overfit to the given grammar, e.g. by hardcoding nonterminals or terminals in your implementation. Your approach should work for arbitrary grammars. We will use a slightly different grammar (that will also produce JSON objects) to evaluate your submission.
- Your approach should be general: It should also work for different grammars, different constraints, and different sample inputs to the ones provided in this project.
- Use the EarleyParser of the FuzzingBook to parse a given input to a tree. You may assume the grammar is unambiguous, and return the first tree returned by the parse function of the EarleyParser.

Framework

The file marked with **[C]** should be changed by you.

The files marked with **[X]** must not be changed.

```
.
├── [X] inputs/
├── [X] helpers.py
├── [C] implementation.py
├── [X] main.py
├── [X] oracles.py
├── [X] printer_grammar.py
├── [X] tests.py
├── [X] verify.py
```

You will find 6 different oracles in **oracles.py**. For each oracle, you will find 1000 passing inputs and 1000 failing inputs in folder `./inputs/oracle_name`.

The grammar that was used to generate these inputs can be found in `printer_grammar.py`.

Your task is to implement all the functions in `implementation.py`. You may add arbitrary code and functions in this file, however you may not change the function signature of any of the functions you should implement. No other file may be added or changed.

The main logic is implemented in **main.py**.

Usage of main.py

```
>
usage: main.py [-h] [--test-instantiate-nonterminals] [--
test-instantiate-subtrees] [--test-check] [--test-learn] [--
test-generate] [--test-all] [--learn-oracle-1] [--learn-
oracle-2] [--learn-oracle-3]
               [--learn-oracle-4] [--learn-oracle-5] [--
learn-oracle-6] [--learn-all]

Project 2

options:
  -h, --help            show this help message and exit

Test Options:
  Options for enabling tests.

  --test-instantiate-nonterminals
                        Do test
test_instantiate_with_nonterminals.
  --test-instantiate-subtrees
                        Do test
test_instantiate_with_subtrees.
  --test-check          Do test check.
  --test-learn          Do test learn.
  --test-generate       Do test generate.
  --test-all            Do all tests.

Main Options:
  Options for the project.

  --learn-oracle-1      Learn the constraints of oracle 1.
  --learn-oracle-2      Learn the constraints of oracle 2.
  --learn-oracle-3      Learn the constraints of oracle 3.
  --learn-oracle-4      Learn the constraints of oracle 4.
  --learn-oracle-5      Learn the constraints of oracle 5.
  --learn-oracle-6      Learn the constraints of oracle 5.
  --learn-all           Learn the constraints of all
oracles.
```

- Make sure that all unit tests run successfully for your implementation by running: `python3 main.py --test-all`.
- Make sure that the constraints of all oracles can be learned with your implementation by running: `python3 main.py --learn-all`.

Project Tasks

Start by learning the constraints for each of the six provided oracles. Your implementation must be general enough to work on four additional unknown oracles. It should automatically learn constraints from any set of passing/failing inputs and produce passing inputs for each oracle using functions `check(...)` and `generate(...)`

You may use any fuzzers provided by FuzzingBook or implemented by yourself in Python. Other (third-party) fuzzers or programming languages are not allowed.

To get you started, find the grammar in **printer_grammar.py** that produces the `printer` object with the following definition:

In [ ]:
```
printer_object = {
  "manufacturer": str,
  "model": str,
  "serialNumber": int,
  "type": str,
  "status": str,
  "copiesPrinted": int,
  "resolution": list,
  "operatingSystem": list,
  "securityFeatures": list,
  "hasWarranty": int,
  "needsCheck": int
}
```

How We Evaluate Your Project

- (1) We will run `python3 main.py --learn-oracle-{i}` where the number `{i}` corresponds to each combination of grammar/passing and failing inputs/oracle, and collect the constraints learned for each oracle (i.e., the return value of `learn_and_refine`).
- (2) We will use a large set of unknown *passing* and *failing* inputs and measure how many of these inputs are correctly classified by the constraints learned by `learn_and_refine`. The number of correctly classifies inputs determines your points.

Notes

- All tests muss pass. That is, if `python3 main.py --test-all` does not pass, you will fail the project.

- For any given oracle, the execution of `python3 main.py --learn-oracle-{i}` should not take more than 3 minutes. Taking longer than 3 minutes for a given oracle means you will get 0 points for that given oracle.
- We will evaluate your submission on a slightly different version of the `printer_grammar` in order to avoid overfitting. Also, we will use similar oracles to the ones provided to you, but not the same ones, to avoid overfitting.

Evaluation Guidelines

Our evaluation process will encompass both the known oracles provided to you and a set of unknown oracles. Here's how we will proceed:

Evaluation with Known Oracles (6 pts)

Points per Oracle: For each of the 6 provided oracles, you will receive +1 point for correctly learning its constraints. This means you can score up to 6/6 points in this section. Correctly learning an oracle means that the constraints output by the `learn_and_refine` function correctly classify all sample inputs in less than 3 minutes.

Evaluation with Unknown Oracles (4 pts)

Additional Oracles: Beyond the 6 known oracles, your implementation will be tested on 4 additional unknown oracles, which you will not have access to beforehand. Successfully handling each of these unknown oracles will grant you +1 point. For these unknown oracles, we will employ a different grammar (not the printer grammar) for generating JSON objects. This new grammar will be structured similarly to the printer grammar, with comparable constraints. Correctly learning the oracle constraints means that at least 95% of the inputs are classified correctly.

Guaranteed passing criterium

If you do not pass by our evaluation guidelines, you are still guaranteed to pass the project if you meet all of the following points:

1. **Pass tests**: All tests muss pass: `python3 main.py --test-all`
2. **Correctly learning 4/6 oracle constraints**: `python3 main.py --learn-oracle-{i}` is successful for at least 4 different oracles, i.e. it outputs a set of constraints that classify passing and failing inputs correctly.
3. **Time limit**: For the oracles that are successfully learned, the execution time of `python3 main.py --learn-oracle-{i}` must not exceed 3 minutes per oracle.

Submission format

Submit your solution as a ZIP file on your status page in the CMS.

`solution.zip` should only contain contain `implementation.py`

Double-check that your submission is runnable. We cannot grade submissions that are not runnable or have bugs.