Project 3 : Coverage-Based Fuzzer

March 16, 2017

1 Part 1: Evolutionary Fuzzing

Implement a coverage-based fuzzer that is able to evolve inputs by using *mutation*. Use coverage as the *fitness* of your inputs. For determining the coverage you can use the `sys.settrace()` function as shown in the lecture for the most basic needs. If you want to come up with more sophisticated guidance criteria, we encourage you to use the `trace` module[1] or the `coverage` package[2]. We recommend that you start from your grammar-based fuzzer from Project 2. As before, the grammars are given in the format specified by the `grammar` subject and are available for all public subjects. And again, your fuzzer must be initialized by the provided random seed to make executions deterministic. We consider all inputs that result in a non-zero exit code of the subject and do not raise a `ParseException` as failure inducing.

In order to invoke the subject from your fuzzer and measure its coverage, we now recommend loading the subject module dynamically from file and calling its `main(filename)` method directly from your code:

```
import importlib.util as iu
import sys

# load the subject module
spec = iu.spec_from_file_location("arithmeticExpr.arithmetic",
                                  "arithmeticExpr/arithmetic.py")
subj = iu.module_from_spec(spec)
# initialize the subject
spec.loader.exec_module(subj)

# now you can execute the subject by calling its main method
subj.main('path/to/your/generated/input.sample')
```

For your algorithm you can get inspiration from the wonderful book on "Essentials of Metaheuristics" available for free at `https://cs.gmu.edu/~sean/book/metaheuristics/`.

Notes on Mutation:

- You should use a derivation-tree-based representation of your inputs for mutation. Do not mutate strings.
- As with generation – make sure to limit the depth of derivation trees to avoid non-termination and exceedingly large inputs.

Your implementation is expected to write exactly one generated input per found failure into the working directory having `.sample` as file ending. You are allowed to create temporary directories for generated samples in order to use them as inputs for the provided subject. At the end of the execution these temporary directories and inputs that are not failure inducing must be deleted. The fuzzer will be graded based on the number of unique exceptions that are triggered in each subject.

2 Part 2: Crossover

The second part of this project is adding the *crossover* of two inputs to your coverage-based fuzzer. Figure 1 illustrates one such crossover of two derivation trees for a simplified version of `URL` grammar. The nodes of your derivation trees should be aware of their corresponding definitions in the grammar such that you can cross-combine only compatible subtrees. This means that you should only be able to

[1] `https://docs.python.org/3.6/library/trace.html`
[2] `https://coverage.readthedocs.io/`

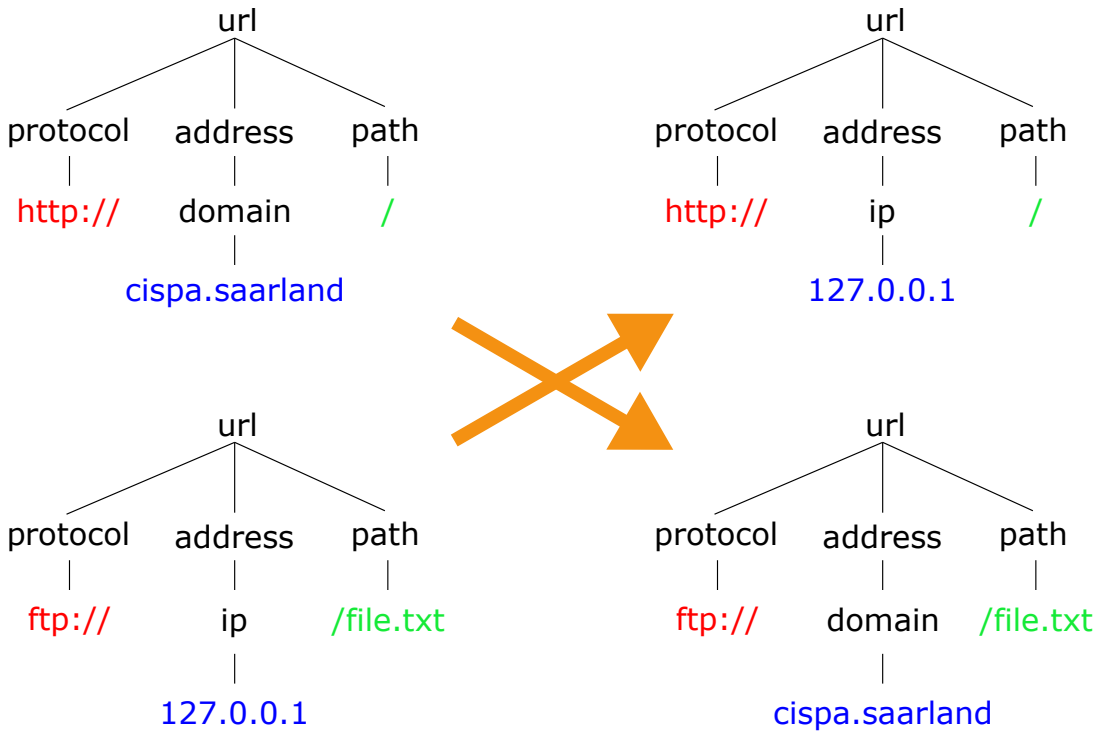

Figure 1: Example Crossover of two Derivation Trees

swap similarly derived parts of the inputs with each other. In the given example, the `ip` and `domain` derivation nodes could be swapped because both of them fit under the `address` node. Similarly, equal nodes can be swapped trivially (e.g. `protocol` nodes among each other). However, swapping a `path` node with a `domain` node should be impossible.

3 Part 3: Combining Coverages

In this part of the project you will combine the feedback from the grammar coverage and program coverage and try to get your fuzzer to produce inputs that have good values for both. Since the program coverage is a value between zero and unknown and the grammar coverage can be expressed as a percentage, we recommend using the latter as a *secondary criterion* for fitness comparisons. This means that the grammar coverage should only be considered if the program coverages of two inputs are equal.

Let us consider the following example, where the fitness is denoted as a pair (program coverage, grammar coverage). According to our recommended comparison the following should hold:

`(21,0.3)` is better than `(20,0.5)` (the primary criterion is better)

`(20,0.8)` is better than `(20,0.5)` (the primary criteria are equal and the secondary criterion is better)

`(20,0.7)` is as good as `(20,0.7)` (both criteria are equal)

For additional experimenting, you could try to include the file size as a *tertiary criterion* and seeing if you can manage to reduce the size of the generated inputs without using an extra minimization step. *(This task is not mandatory and serves only as a suggestion.)*

4 Implementation

The subjects for the course projects are hosted as a public project on our Gitlab `https://securitytesting.cispa.saarland/kampmann/subjects`. Please make sure to pull the most recent revision of the subjects

from the project (*especially since we added the `main(filename)` functions*). Each subject resides in a top level directory and you can invoke it using:

```
python <subject>/<module> <inputfile>
```

Depending on your setup you might need to substitute `python` for `python3` if your system uses Python 2.x by default. The subjects will terminate with a non-zero exit code in case of an error. Additionally to the public subjects we will also evaluate your implementation on two secret subjects and on variants of the public subjects that contain additional bugs. Your implementation is expected to be accessible as a python module `coveragefuzzer.py` in the root directory of your project repository. The fuzzer is supposed to be invokable as:

```
python coveragefuzzer.py -s <seed> -t <timeout> -g <grammar>
                         -p <path-to-subject> -m <name-of-subject-module>
```

The `m` and `p` parameters are intended to be passed to the `spec_from_file_location` function.

The produced failure inducing samples should be written to the working directory using `.sample` as file ending. Like in the previous projects the timeout is given in seconds.

In order to evaluate your implementation we will use 5 seeds provided by you in a text file `seeds.txt` stored in the root directory of your project repository containing one seed per line. Additionally to the seeds provided by you, we will choose 5 additional random seeds. For each seed we will run your implementation with a timeout of 5 minutes.