CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Security Testing

WS 2021/2022

Prof. Dr. Andreas Zeller
Leon Bettscheider
Marius Smytzek

Project 2

Due: 27. February 2022

The lecture is based on The Fuzzing Book (https://fuzzingbook.org/), 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 project 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.

Project Description

In this project, you will extend the implementation of FuzzingBook's **ConcolicTracer** class.

To get started, please read the **ConcolicFuzzer** chapter (https://www.fuzzingbook.org/html/ConcolicFuzzer.html (https://www.fuzzingbook.org/html/ConcolicFuzzer.html)).

Note: Make sure you have the latest fuzzingbook package installed. To upgrade, use

`pip3 install fuzzingbook --upgrade`

Task 1: Random Solving

Implementation

In this task, you should subclass the **ConcolicTracer** class and extend it with a **reval** function. As base for your implementation, use the file `RandomConcolicTracer.py`. Similar to the **zeval** function which is already implemented in the super class, this function should try to find an assignment for the input variables that satisfy the path constraints. However, while **zeval** relies on SMT solving for this purpose, the function you will implement repeatedly chooses variable assignments at random and checks whether they satisfy the path constraints.

The **reval** function should have the following signature:

`def reval(self, attempts: int):`

The `attempts` parameter describes the number of solving attempts before giving up and returning that the path constraints cannot be satisfied (*unsat*).

The return value format of **reval** should be identical to that of **zeval** with argument `python=False`.

You will need to implement random generation of the datatypes *Int* and *String*. You must use the random generation functions *random_int* and *random_string* defined in `RandomGeneration.py`. This file must not be changed.

Hints:

- The path constraints are stored in the class variable `path`.
- The declared variables are stored in the class variable `decls`.
- To evaluate a given python expression, you can use the built-in *eval* function (https://docs.python.org/3/library/functions.html#eval (https://docs.python.org/3/library/functions.html#eval)).

Evaluation

Now, benchmark your **reval** function with the **zeval** function on a set of 3 sample programs stored at `samples/`. The main purpose of this part of the task is to provide you with a way to check your implementation.

To run the time benchmark, execute:

`python3 Benchmark.py`

Make sure that the benchmark passes, and hand in your implementation `RandomConcolicTracer.py` and the full output of the benchmark at `benchmark_result.txt`.

Note:

- During grading, we will use *slightly* different sample programs, to avoid overspecialized solvers.

Task 2: Function summaries

Background

The implementation of FuzzingBook's **ConcolicTracer** class does not currently provide *function summaries* for all operations on native types. With a function summary, we can *summarize* a function by expressing a postcondition on the function output given the input. The main purpose of this is to mitigate the path explosion problem, which states that the number of possible execution paths in a program grows exponentially with the program size. By using a function summary, we can directly fall back to the output postcondition, which eliminates the need to execute the function symbolically (which would mean to explore many execution paths). Additionally, in Python, some builtin functions are implemented in C. This makes the implementation of a symbolic execution engine difficult, as it would have to switch between Python and C code. In this context function summaries can help too.

In this part of the project, you will implement a number of *function summaries* that mimick the targeted function's behavior symbolically.

Implementing a function summary

When implementing a function summary, the first step is to think about what the summarized functions should do: Which properties must hold on the input for the function to return a particular output? Next, how can you encode these constraints in z3's language?

We recommend to use z3's Python API. As a starting point, have a look at the interface description at https://z3prover.github.io/api/html/namespacez3py.html (https://z3prover.github.io/api/html/namespacez3py.html) . It can also help to read z3 code snippets e.g. on StackOverflow.

Example: Summarizing `str.isalpha`

The builtin `str.isalpha` method is specified as follows by the docs at https://docs.python.org/3/library/stdtypes.html#str.isalpha (https://docs.python.org/3/library/stdtypes.html#str.isalpha) :

Return True if all characters in the string are alphabetic and there is at least one character, False otherwise. Alphabetic characters are those characters defined in the Unicode character database as "Letter" [...]

Have a look at the following input-output examples:

```
>>> 'aaa'.isalpha()
True
>>> 'a?a'.isalpha()
False
>>> '123'.isalpha()
False
>>> 'abc def'.isalpha()
False
```

To simplify matters, we restrict ourselves to ASCII rather than Unicode in the function summary implementations.

Note that there can be multiple ways to implement a function summary, with different implications on the run time.

For instance, the following two function summary implementations for `str.isalpha` produce the same results, but the second one is usually faster:

Implementation 1: z3.Contains and negation

```
class zstr(zstr):
    def isalpha(self):
        non_alphas = []
        for i in range(256):
            if not ((i >= ord('a') and i <= ord('z')) or (i >= ord('A') and i <=
ord('Z'))):
                non_alphas.append(z3.Unit(z3.BitVecVal(i, 8)))

        name = 'isalpha_%d' % fresh_name()
        concrete_result = self.v.isalpha()
        result = zbool.create(self.context, name, concrete_result)
        sym_constraint = z3.And(z3.Length(self.z)>0, z3.Not(z3.Or([z3.Contains(self.z,
na) for na in non_alphas])))
        self.context[1].append(sym_constraint == result.z)
        return result
```

Implementation 2: z3.InRe

```
class zstr(zstr):
    def isalpha(self):
        charset = z3.Union([z3.Re(z3.StringVal(c)) for c in string.ascii_letters])
        template = z3.Star(charset)

        name = 'isalpha_%d' % fresh_name()
        concrete_result = self.v.isalpha()
        result = zbool.create(self.context, name, concrete_result)
        sym_constraint = z3.And(z3.Length(self.z)>0, z3.InRe(self.z, template))
        self.context[1].append(sym_constraint == result.z)
        return result
```

Goal 1 (70%)

The first and main goal is to test a checker for e-mail addresses (stored at `EMail.py`) using concolic execution. This checker program is also shown below.

Given the `addr` string input, the function checks whether the input conforms to a certain *e-mail address format*. More precisely, this means that, among other constraints, the input must contain a `@` sign, and should end on `.com`, `.de`, or `.fr`.

To test this program, concolic execution should generate a set of inputs, such that executing all these inputs achieves **100%** code coverage in the `check_mail` function.

This function cannot be tested using fuzzingbook's concolic fuzzer out-of-the-box, as it relies on Python builtin functions that are not yet summarized: `str.isupper`, `str.endswith` and `str.__contains__`. As an exception, a function summary for `str.find` is already implemented in the fuzzingbook, and does not need to be implemented in this project.

```
def check_mail(addr: str):
    print('Input to check_mail: ', addr)

    if addr.find('.') == 10:
        if addr.find('@') == 3:
            prefix = addr[:3]
```

```python
                if prefix.isupper():
                    if addr.endswith('.com'):
                        if 'wow' in addr:
                            print('Accepted! (com)')
                            return True
                        else:
                            print('Not accepted! wow not found. (com)')
                            return False
                    elif addr.endswith('.de'):
                        if 'such' in addr:
                            print('Accepted! (de)')
                            return True
                        else:
                            print('Not accepted! such not found. (de)')
                            return False
                    elif addr.endswith('.fr'):
                        if 'fuzz' in addr:
                            print('Accepted! (fr)')
                            return True
                        else:
                            print('Not accepted! fuzz not found. (fr)')
                            return False
                    else:
                        print('Not accepted! (invalid)')
                        return False
                else:
                    print('Not accepted! (prefix not upper)')
                    return False
        else:
            print('Not accepted! (@ not in expected range)')
            return False
    else:
        print('Not accepted (. not at expected position)')
        return False
```

Your task is to implement the following function summaries in `Summaries.py`:

- `str.__contains__` with signature:

  ```python
  def __contains__(self, m: str) -> zbool:
  ```

- `str.endswith` with signature:

  ```python
  def endswith(self, other: zstr, start: int = None, stop: int = None) -> zbool:
  ```

- `str.isupper` with signature:

  ```python
  def isupper(self) -> zbool:
  ```

Note: For `str.isupper`, the summary is allowed to return `True` if all characters in the string are in `string.ascii_uppercase`. For `str.endswith` you should assume that the `start` and `stop` parameters are always `None`.

The steps in this task are:

- You should implement your function summaries in `Summaries.py`.
- Then, run `ConcolicEMailExploration.py`, which will generate 200 inputs for the e-mail program using concolic execution, and serialize the inputs to `inputs.json`.
- To replay the generated inputs in `EMail.py` you can use the `RunInputs.py` file.
- If all summaries were implemented correctly, the generated inputs should achieve **100%** statement coverage in `EMail.py`.

To summarize:

```
# Generate and serialize inputs
python3 ConcolicEmailExploration.py

# Replay inputs in EMail.py and measure coverage
python3 -m coverage run RunInputs.py

# Generate a coverage report
python3 -m coverage report EMail.py
```

Ideally, you should get the following output:

```
Name        Stmts   Miss  Cover
-------------------------------
EMail.py       32      0   100%
-------------------------------
TOTAL          32      0   100%
```

Hints:

- To draw some inspiration, have a look at the implementation of the `startswith` , `upper` , `find` , `lower` , `find` , `lstrip` , and `rstrip` function summaries. To read the code of these functions, head over to https://www.fuzzingbook.org/html/ConcolicFuzzer.html#A-Concolic-Tracer (https://www.fuzzingbook.org/html/ConcolicFuzzer.html#A-Concolic-Tracer) and expand the `Implementing ConcolicTracer` section.
- The builtin `str.__contains__` method is used to check whether a string `needle` is contained in another string `haystack` . It is implicitely invoked by Python on checks such as `if 'needle' in 'haystackneedlehaystack':` .

Goal 2 (30%)

The second goal is to implement additional function summaries in `Summaries.py` .

To test whether your implementations work as intended, we provide tests that can be run by executing `TestSummaries.py` .

In total, there are 12 summaries that should be implemented. One summary operates on the `int` type, whereas the other 11 operate on the `str` type.

The int summary

- `zint.__abs__`

  ```
  def __abs__(self): -> zint
  ```

The builtin `__abs__` method returns the absolute value for a number. For more details, please refer to https://docs.python.org/3/library/functions.html#abs (https://docs.python.org/3/library/functions.html#abs) . You should only implement this method for the `int` (i.e. `zint`) datatype.

Have a look at the following input-output examples:

```
>>> abs(0)
0
>>> abs(42)
42
>>> abs(-42)
42
```

You should implement the method `__abs__` in file `Summaries.py` .

String summaries

You should implement function summaries for the following built-in string functions. For more information on these functions, you may refer to https://docs.python.org/3.9/library/stdtypes.html#string-methods (https://docs.python.org/3.9/library/stdtypes.html#string-methods) .

- `str.capitalize`

  ```
  def capitalize(self) -> zstr:
  ```

- `str.isalnum`

  ```
  def isalnum(self) -> zbool:
  ```

- `str.isdecimal`

  ```
  def isdecimal(self) -> zbool:
  ```

- `str.isdigit`

```python
def isdigit(self) -> zbool:
```

- `str.islower`

```python
def islower(self) -> zbool:
```

- `str.isnumeric`

```python
def isnumeric(self) -> zbool:
```

- `str.isprintable`

```python
def isprintable(self) -> zbool:
```

- `str.isspace`

```python
def isspace(self) -> zbool:
```

- `str.rfind`

```python
# You may assume that start and stop are always None.
def rfind(self, sub: str, start:int = None, stop:int = None) -> zint:
```

- `str.swapcase`

```python
def swapcase(self) -> zstr:
```

- `str.title`

```python
def title(self) -> zstr:
```

Note:

- We assume that strings only consists of **ASCII** characters. In particular, you do not have to model Unicode characters.
- If the function signature we provide declares a parameter with a native type (e.g. int instead of zint), you may assume this parameter to be concrete.
- During grading, we will use *slightly* different tests to check your function summaries, to avoid overspecializations.

Notes

- You may work on this project in groups of two people. You're also allowed to work on the project individually. We ask you to make a decision via your Personal Status page until 30.01.2022.

Evaluation Guidelines

To pass the project, you will need to achieve at least half the points in both Task 1 and Task 2. The tasks are weighted as follows:

Task 1: 33.3% of total points

Task 2: 66.6% of total points

Guaranteed passing criterium

You are guaranteed to pass this project if you meet `Goal 1` of `Task 2`, i.e., the function summaries you have implemented achieve **100%** statement coverage in `EMail.py`.