CISPA
HELMHOLTZ CENTER FOR
INFORMATION SECURITY

Automated Debugging

WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Exercise 7 (10 Points)

Due: 27. January 2023

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 must not delete any provided ones. You can verify whether your submission is valid by calling python3.

```
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 7-1: No Silver Bullet (5 Points)

Delta debugging and abstract failure-inducing input are powerful in many scenarios. But you can always find scenarios where these two technologies become not so helpful. One example is heartbeat (`exercise_1.py`), which we have seen several times in the previous exercises:

In [1]:
```python
data = 'password:hjasdiebk456jhaccount:smytzek'

def store_data(payload: str):
    global data
    data = payload + data

def get_data(length: int) -> str:
    return data[:length]

def heartbeat(length: int, payload: str) -> str:
    assert length == len(payload)
    store_data(payload)
    assert data.startswith(payload)
    r = get_data(length)
    assert r == payload
    return r
```

To better understand why these two technologies become less helpful on the heartbeat example, let's conduct an experiment as follows:

1. Build a grammar for the `payload` argument that can be of arbitrary length.
2. Use the `GrammarFuzzer` introduced in the lecture to generate a random input for the `heartbeat` function (you may fix the `length` argument, say fix it to 5, to avoid fuzzing integers) so that it fails (i.e., to violate the assertions).
3. Apply `DeltaDebugger` on the random failure input you found: What minimized input do you get? How well does it explain the failure reason?
4. Apply `DDSetDebugger` on the random failure input you found: What generalized pattern do you get? Based on this pattern:

- 4.1. Do you think the pattern is correct, that is, all of its instantiations are failure inputs to `heartbeat`?
- 4.2. (Now, to examine your thoughts on 4.1) Invoke the `fuzz_args` method of `DDSetDebugger` to produce 10,000 instantiations of the generalized pattern, calculate how many of them are failure inputs to `heartbeat` -- let's call this rate the *success rate*. Is the success rate high?
- 4.3. To enhance the informativity/expressiveness of `DDSetDebugger` -- so that is becomes more helpful for the `heartbeat` case -- what do you think can be a potential improvement or extension?

Please submit a **report** that describes how you conduct the experiment following the above steps. Specifically, your report should include:

- The Python code (including the grammar of step 1) you use for the experiment.
- The outputs of your code: the random input (step 2), the minimized input (step 3), the generalized pattern and the success rate (step 4).
- Your answers to the questions mentioned above.
- Optionally, any explanation to any piece of hard-to-understand code/function.

The report can be in **either** format:

- A markdown document `exercise_1.md` consisting of code and text (for your outputs, answers and additional explanations).
- Or, a Jupyter notebook (if you learned how to use it -- but not required for this course) consisting of executable code, running outputs, and markdown blocks for the answers and additional explanations. Remember to rename `exercise_1.md` to `exercise_1.ipynb`.

Keep your report clear, structured (with titles numbering the steps), easy-to-follow, and even better -- simpler and shorter. Misunderstanding may leads to low scores for this exercise.

Exercise 7-2: Semantic-Preserving Program Slicing (5 Points)

As mentioned in the book, one particularly fun application of delta debugging is to reduce program code -- a fancier terminology for this operation is *program slicing*. You learned flow-analysis-based slicing in this chapter. In this exercise, you are going to investigate another kind of program slicing via a combination of AST-based delta debugging and testing.

Recall in the delta debugging chapter, this section introduces how to reduce program code on AST nodes so that we can find a minimized program that triggers the failures (i.e., failed tests). Thinking oppositely, we could also find a minimized program that **passes** all the tests is found, and this minimized program can be regarded approximately semantically equivalent to the original program since they have the **same behaviors** according to the tests. The minimized program is thus regarded as the semantic-preserving slice.

To implement such a slicing functionality, we can reuse the `DeltaDebugger`, in a similar way to what we learned from the lecture:

```
with DeltaDebugger() as dd:
    compile_and_test_ast(fun_tree, fun_nodes, test_tree)
```

But here, the `compile_and_test_ast` function should execute a piece of code (manually constructed from AST) that only fails when: all the tests are **passed**, i.e., the function represented by `test_tree` does not raise any exception. In this way, the delta debugger will be able to find a minimized program that still fails `compile_and_test_ast` -- but, this is equivalent to the situation where all the tests defined in `test_tree` get passed. Finish your implementation of this function in `exercise_2.py`:

```
def compile_and_test_ast(tree: ast.Module, keep_list: List[ast.AST],
                         test_tree: ast.FunctionDef) -> None:
    pass # YOUR CODE HERE
```

As an example, you can test your `compile_and_test_ast` on slicing the Fibonacci function:

```
In [7]: def fib(n: int) -> int:
            if n == 0 or n == 1:
                return 1
            return fib(n - 1) + fib(n - 2)

        def fib_test_simple():
```

```python
    assert fib(0) == 1
    assert fib(1) == 1

fib_test_simple()
```

First, create the input ASTs:

In [8]:
```python
import inspect, ast
from debuggingbook.DeltaDebugger import NodeCollector, DeltaDebugger
from debuggingbook.bookutils import print_content

fun_tree: ast.Module = ast.parse(inspect.getsource(fib))
fun_nodes = NodeCollector().collect(fun_tree)
test_tree: ast.Module = ast.parse(inspect.getsource(fib_test_simple)).body[0]
```

Then, apply the `DeltaDebugger`:

```python
with DeltaDebugger() as dd:
    compile_and_test_ast(fun_tree, fun_nodes, test_tree)

reduced_nodes = dd.min_args()['keep_list']
reduced_fun_tree = copy_and_reduce(fun_tree, reduced_nodes)
print_content(ast.unparse(reduced_fun_tree), '.py')
```

which finds the following slice:

```python
def fib(n):
    return 1
```

This makes sense as the test `fib_test_simple` only covers the cases when `n == 0 or n == 1`. If you add another testcase, say `assert fib(4) == 5`, then all branches of `fib` is covered so that nothing can be reduced for the `fib` function.