CISPA
HELMHOLTZ CENTER FOR INFORMATION SECURITY

Automated Debugging
WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Project 1 (100 Points)

Due: 29. January 2023

The lecture is based on The Debugging Book, an *interactive textbook that allows you to try out code right in your web browser*.
The Debugging Book code is additionally available as a Python pip package. To work on the project, please install the package locally:

```
pip3 install debuggingbook
```

Submit your solutions as a Zip file on your status page in the CMS.

We will provide you with 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 verify.py
```

The output provides an overview of whether a required file, variable, or function is missing and if you altered a function's pattern. We can only evaluate your submission if you follow this structure. A non-evaluable project will result in 0 points, so verify your work before submitting it. Note that the script only reveals if you have solutions we can evaluate.

Please use Python 3.10 to work on this project. We will use Python 3.10 for the evluation so make sure your code runs under this version of Python.

Please read the entire project description carefully before starting to implement.

How to Pass the Project

This project has two tasks. In the first task, you will combine slicing with statistical debugging to correlate dependencies with the occurrence of failures. In the second task, you will implement your fault localization in the manner you like.

You can achieve a maximum of 100 points for this project; on the first task, you get up to 60 points, and on the second, up to 40 points.

Passing this project depends on a minimum passing criterium. To complete this criterium, you must pass all tests in **slicer_statistical_debugging_tests.py**. This achievement will grant you 50 points for task 1 besides passing the project.

The Subjects of the Project

The project provides four subjects you can find and investigate in the **subjects** directory. Each of these subjects comes with some tests. These subjects are:

- **subjects/middle**: The `middle` function of the Debugging Book. It contains the same fault as the example in the book.
- **subjects/remove_html_markup**: The `remove_html_markup` function of the Debugging Book. It contains the same fault as the example in the book.
- **subjects/sqrt**: An unasserted implementation of a square root function. Giving that function a negative number raises a `ZeroDevisionError`.
- **subjects/bf**: A simplified brainf*ck interpreter that does not support loops. The fault of this interpreter lies in the parsing, where it adds a wrong token for the `-` symbol.

Task 1-1: Fault Localization with Slices (60 Points)

How Are the Points Distributed

As described above, you will receive 50 points for passing the public tests in **slicer_statistical_debugging_tests.py**. You will receive the remaining 10 points based on the number of secret tests you may pass, e.g., if you are successful on 50% of our secret tests, you will receive 5 points.

Where to Implement this Task

Implement your solution for *Task 1-1* in **slicer_statistical_debugging.py**.

How to Evaluate this Task

We also provide you with tests for checking your implementation. You can run tests by leveraging Python's `unittest` module. You can run tests for *Task 1-1* by executing the following on your command line:

```
python3.10 -m unittest slicer_statistical_debugging_tests
```

You can also run certain test classes or functions by specifying them as follows:

```
python3.10 -m unittest slicer_statistical_debugging_tests.InstrumentTests

python3.10 -m unittest \
    slicer_statistical_debugging_tests.InstrumentTests.test_structure_middle
```

Description

In this task, you will combine slicing with the techniques from statistical fault localization. The events for the statistical debugging are the dependencies extracted from the Slicer.

Consider the middle function from the Debugging Book.

In [1]:
```
def middle(x, y, z):
    if y < z:
        if x < y:
            return y
        elif x < z:
            return y
    else:
        if x > y:
```

```
            return y
        elif x > z:
            return x
    return z
```

We need a test function to verify our results.

In [2]:
```
def test_middle(x, y, z, expected):
    return middle(x, y, z) == expected
```

Now we can run some tests and check the results.

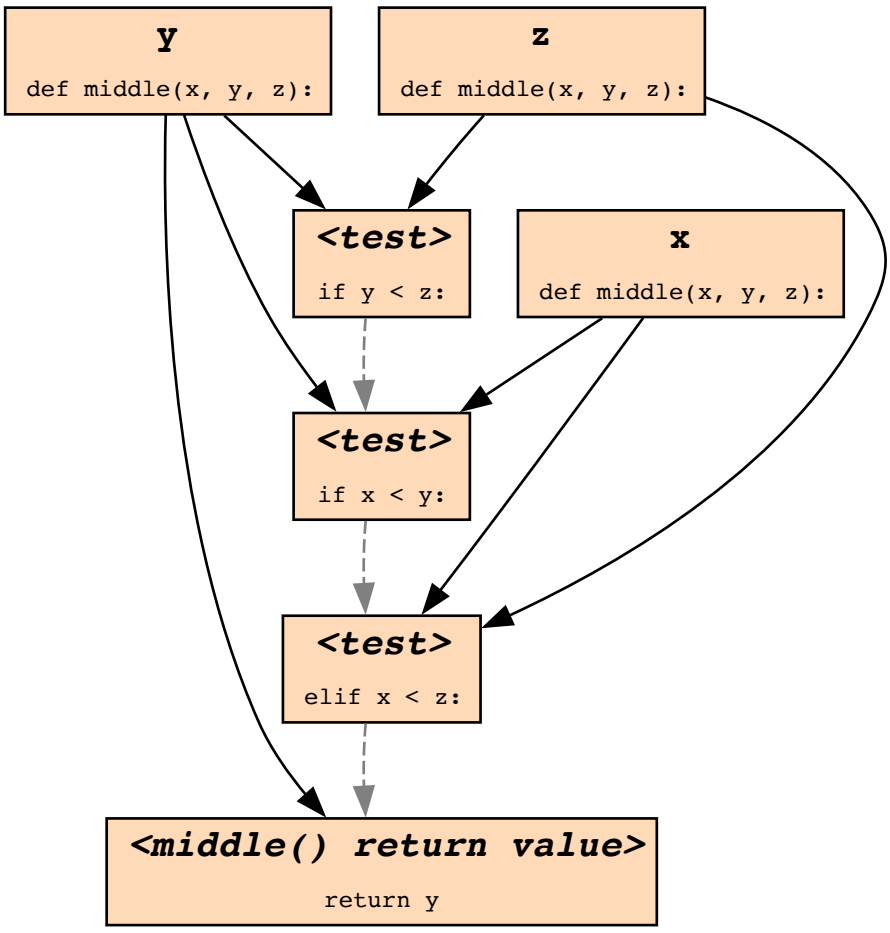
In [3]:
```
results = [
    test_middle(x=3, y=3, z=5, expected=3),
    test_middle(x=1, y=2, z=3, expected=2),
    test_middle(x=3, y=2, z=1, expected=2),
    test_middle(x=5, y=5, z=5, expected=5),
    test_middle(x=5, y=3, z=4, expected=4),
    test_middle(x=2, y=1, z=3, expected=2),
]

print(f'Passed {sum(results)} of 6 tests')
```

```
Passed 5 of 6 tests
```

With our Slicer, we can extract the `middle()` dependencies.

In [5]:
```
with Slicer(middle) as slicer:
    test_middle(x=3, y=3, z=5, expected=3)
slicer
```

Out[5]:



But we can also get the dependencies in a more machine-readable format.

For the data dependencies, we can do the following:

```
In [6]: slicer.dependencies().data
```

```
Out[6]: {('x', (<function __main__.middle(x, y, z)>, 1)): set(),
 ('y', (<function __main__.middle(x, y, z)>, 1)): set(),
 ('z', (<function __main__.middle(x, y, z)>, 1)): set(),
 ('<test>',
  (<function __main__.middle(x, y, z)>, 2)): {('y',
   (<function __main__.middle(x, y, z)>, 1)), ('z',
   (<function __main__.middle(x, y, z)>, 1))},
 ('<test>',
  (<function __main__.middle(x, y, z)>, 3)): {('x',
   (<function __main__.middle(x, y, z)>, 1)), ('y',
   (<function __main__.middle(x, y, z)>, 1))},
 ('<test>',
  (<function __main__.middle(x, y, z)>, 5)): {('x',
   (<function __main__.middle(x, y, z)>, 1)), ('z',
   (<function __main__.middle(x, y, z)>, 1))},
 ('<middle() return value>',
  (<function __main__.middle(x, y, z)>, 6)): {('y',
   (<function __main__.middle(x, y, z)>, 1))}}
```

For the control dependencies, we need to access the `control` attribute:

```
In [7]: slicer.dependencies().control
```

```
Out[7]: {('x', (<function __main__.middle(x, y, z)>, 1)): set(),
 ('y', (<function __main__.middle(x, y, z)>, 1)): set(),
 ('z', (<function __main__.middle(x, y, z)>, 1)): set(),
 ('<test>', (<function __main__.middle(x, y, z)>, 2)): set(),
 ('<test>',
  (<function __main__.middle(x, y, z)>, 3)): {('<test>',
   (<function __main__.middle(x, y, z)>, 2))},
 ('<test>',
  (<function __main__.middle(x, y, z)>, 5)): {('<test>',
   (<function __main__.middle(x, y, z)>, 3))},
 ('<middle() return value>',
  (<function __main__.middle(x, y, z)>, 6)): {('<test>',
   (<function __main__.middle(x, y, z)>, 5))}}
```

In this task, you should investigate both data and control dependencies.

Your goal in this task is to implement a technique that finds the correlation of specific dependencies with the occurrence of faults in the same manner as statistical debugging correlates, for instance, lines with failures.

As an example, you need to find how the data dependency `<middle() return value> ← y` (`<middle() return value>` depends on the variable `y`) correlates with the fault. This dependency is represented by `('<middle() return value>', (<function __main__.middle(x, y, z)>, 6)): {('y', (<function __main__.middle(x, y, z)>, 1))}` in our `slicer.dependencies().data`. As in statistical debugging, this correlation means how often it occurs in faulty/passing runs, does it only occur in faulty runs, and how suspicious the data dependency is. The same holds for instance for the control dependency `if x < y ← if y < z`, represented by `('<test>', (<function __main__.middle(x, y, z)>, 3)): {('<test>', (<function __main__.middle(x, y, z)>, 2))}` in our `slicer.dependencies().control`.

a. Instrumentation

The first step is to implement an instrumentation of entire Python projects in the `Instrumenter` class. The `Instrumenter` has a method `instrument(self, source_directory: Path, dest_directory: Path, excluded_paths: List[Path], log=False) -> None` that serves as the main access point to your instrumentation. The parameters of this method are the following:

- `source_directory: Path` : The source directory of the project you should instrument.
- `dest_directory: Path` : The destination directory of your instrumented project.
- `excluded_paths: List[Path]` : A list of excluded paths you do not need to instrument.

Your instrumentation should iterate over the files and subdirectories in the source directory and copy all files to the destination directory. During this process, you must find all `python` files you should instrument. You should ignore all paths in the excluded ones from the instrumentation, but you should still copy all excluded files and directories to the destination.

For the instrumentation of python files, you need to implement all appropriate methods in the `Instrumenter` class that are part of the `NodeTransformer`, such that you produce the same transformations as the `Slicer` instrumentation from the Debugging Book. Remember that importing and leveraging any existing part of the Debugging Book is allowed. You are also entitled to change the current code of the `Instrumenter`, but the access method `instrument(self, source_directory: Path, dest_directory: Path, excluded_paths: List[Path], log=False) -> None` must remain unchanged with this exact signature.

For convenience, we will provide you the **lib.py** that will get copied to the destination directory. This library is the access point to the `_data` variable that stores the global `DependcyTracker`, so you should import it in every instrumented file by putting `from lib import _data` at the beginning of the instrumented file. This library also dumps the dependencies after the execution of the project, e.g., a test so that we can use it later during the collection of events.

Overall, your `Instrumenter` should be applied as follows:

In [9]:
```
instrumenter = Instrumenter()
instrumenter.instrument(
    source_directory=Path('subjects', 'middle'),
    dest_directory=Path('tmp'),
    excluded_paths=[Path('subjects', 'middle', 'test_middle.py')]
)
```

A sample result for the instrumented **middle.py** file of the middle subject is shown below:

```
from lib import _data
def middle(x, y, z):
  _data.param('x', x, pos=1)
  _data.param('y', y, pos=2)
  _data.param('z', z, pos=3, last=True)
  if _data.test(_data.get('y', y) < _data.get('z', z)):
    with _data:
      if _data.test(_data.get('x', x) < _data.get('y', y)):
        with _data:
          return _data.set('<middle() return value>', _data.get('y', y))
      else:
        with _data:
          if _data.test(_data.get('x', x) < _data.get('z', z)):
            with _data:
              return _data.set('<middle() return value>', _data.get('y', y))
  else:
    with _data:
      if _data.test(_data.get('x', x) > _data.get('y', y)):
```

```
        with _data:
          return _data.set('<middle() return value>', _data.get('y', y))
        else:
          with _data:
            if _data.test(_data.get('x', x) > _data.get('z', z)):
              with _data:
                return _data.set('<middle() return value>', _data.get('x', x))
  return _data.set('<middle() return value>', _data.get('z', z))
```

However, since **test_middle.py** is excluded, it should not be instrumented but only copied. So it should still look like this:

```
import unittest
from middle import middle


class MiddleTests(unittest.TestCase):

  def test_335(self):
    self.assertEqual(3, middle(x=3, y=3, z=5))

  def test_123(self):
    self.assertEqual(2, middle(x=1, y=2, z=3))

  def test_321(self):
    self.assertEqual(2, middle(x=3, y=2, z=1))

  def test_555(self):
    self.assertEqual(5, middle(x=5, y=5, z=5))

  def test_534(self):
    self.assertEqual(4, middle(x=5, y=3, z=4))

  def test_213(self):
    self.assertEqual(2, middle(x=2, y=1, z=3))
```

b. Collect Dependencies

After executing the instrumented project, the provided `lib.py` library will produce a dump of the events with the following format:

```
(
    ('name of the dependency', ('function_name', line_number)),
    (
        ('name of the dependency', ('function_name', line_number)),
        ('name of the dependency', ('function_name', line_number)),
        ...
    )
)
```

You can access the `_data` variable of this `lib.py` library, which stores the `DependencyTracker`. When the program exits, this library iterates over the dependencies, stores them in a dictionary and uses Python's `pickle` module to write a byte repräsentation of this dictionary to the file **dump**.

For instance, the data dependency `<middle() return value> ← y` from above would then become the following:

```
(
    ('<middle() return value>', ('middle', 6)),
    (
        ('y', ('middle', 1))
    )
)
```

The dump stores all dependencies as a set in a dictionary with the keys `'data'` and `'control'`, that you can access.

You have to implement the `collect(self, DependencyDict)` (do not hesitate because of this type, it just describes the above specification) and `events(self) -> Set[Any]` methods of the `DependencyCollector` class to collect and output the events. The `collect` method gets the loaded dump, i.e., the dictionary with the keys `'data'` and `'control'` that map to a set of dependencies in the specification from above and needs to store them, such that the `events` method will return a set of dependencies in the same format. Feel free to add new methods and members to this class.

After implementing these methods, you can try the debugger as follows:

In [10]:
```
debugger = DependencyDebugger()
```

We can execute the first test:

In [11]:
```
os.chdir(Path('tmp'))
with debugger.collect_pass(Path('dump')):
    subprocess.run(['python3.10', '-m', 'unittest', 'test_middle.MiddleTests
os.chdir(Path('..'))
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

After this passing run, we can now check the collected dependencies:

In [12]:
```
debugger.all_events()
```

Out[12]:
```
{(('<middle() return value>', ('middle', 10)), (('<test>', ('middle',
8)),)),
 (('<middle() return value>', ('middle', 10)), (('y', ('middle', 4)),)),
 (('<test>', ('middle', 6)), ()),
 (('<test>', ('middle', 6)), (('y', ('middle', 4)), ('z', ('middle', 5)))),
 (('<test>', ('middle', 8)), (('<test>', ('middle', 6)),)),
 (('<test>', ('middle', 8)), (('x', ('middle', 3)), ('y', ('middle', 4)))),
 (('x', ('middle', 3)), ()),
 (('y', ('middle', 4)), ()),
 (('z', ('middle', 5)), ())}
```

Note that the line number differs from the one the slicer collected in the beginning. This inconvenience exists because our instrumentation does not keep track of the source and can not connect the actual line number to the instrumented line. You do not need to handle this now so you can ignore it for *Task 1-1*.

Now we can also execute a failing test:

In [13]:
```
os.chdir(Path('tmp'))
with debugger.collect_fail(Path('dump')):
    subprocess.run(['python3.10', '-m', 'unittest', 'test_middle.MiddleTests
os.chdir(Path('..'))
```

Then we can again check the changed dependencies:

In [14]:
```
debugger.all_events()
```

Out[14]:
```
{(('<middle() return value>', ('middle', 10)), (('<test>', ('middle',
8)),)),
 (('<middle() return value>', ('middle', 10)), (('y', ('middle', 4)),)),
 (('<middle() return value>', ('middle', 15)), (('<test>', ('middle', 1
3)),)),
 (('<middle() return value>', ('middle', 15)), (('y', ('middle', 4)),)),
 (('<test>', ('middle', 6)), ()),
 (('<test>', ('middle', 6)), (('y', ('middle', 4)), ('z', ('middle', 5)))),
 (('<test>', ('middle', 8)), (('<test>', ('middle', 6)),)),
 (('<test>', ('middle', 8)), (('x', ('middle', 3)), ('y', ('middle', 4)))),
 (('<test>', ('middle', 13)), (('<test>', ('middle', 8)),)),
 (('<test>', ('middle', 13)), (('x', ('middle', 3)), ('z', ('middle',
5)))),
 (('x', ('middle', 3)), ()),
 (('y', ('middle', 4)), ()),
 (('z', ('middle', 5)), ())}
```

We can also leverage the existing structure to show only the dependencies that occur on a failure:

In [15]:
```
debugger.only_fail_events()
```

Out[15]:
```
{(('<middle() return value>', ('middle', 15)), (('<test>', ('middle', 1
3)),)),
 (('<middle() return value>', ('middle', 15)), (('y', ('middle', 4)),)),
 (('<test>', ('middle', 13)), (('<test>', ('middle', 8)),)),
 (('<test>', ('middle', 13)), (('x', ('middle', 3)), ('z', ('middle',
5)))),)}
```

c. Convert Dependencies to Coverage

There is one final step to do for this task. For this last step, you should extract locations from the dependencies you can then leverage to localize faults.

To accomplish this, you should implement the `def events(self) -> Set[Any]` method of the `CoverageDependencyCollector`. This collector inherits from your already implemented `DependencyCollector`, so you can leverage your implemented structure.

This method should return a set of `('function_name', line_number)` that contains all found locations in the dependencies.

To use it, we create a `DependencyDebugger` with the `coverage` flag set to `True` and rerun the tests.

In [16]:
```
debugger = DependencyDebugger(coverage=True)

os.chdir(Path('tmp'))
with debugger.collect_pass(Path('dump')):
    subprocess.run(['python3.10', '-m', 'unittest', 'test_middle.MiddleTests
with debugger.collect_fail(Path('dump')):
    subprocess.run(['python3.10', '-m', 'unittest', 'test_middle.MiddleTests
os.chdir(Path('..'))
```

We will retrieve the function names and line numbers when getting the events.

In [17]:
```
debugger.only_fail_events()
```

Out[17]:
```
{('middle', 13), ('middle', 15)}
```

Task 1-2: Your Fault Localization (40 Points)

How Are the Points Distributed

You will receive 40 points for this task based on how well your implementation will find the faulty line of code. You can assume that all our secret tests only have a single incorrect line. If your fault localization finds this line as the first ranked location, you will receive all 40 points; if your localization finds the fault in the first ten ranked lines, you will receive fewer points. The better your approach finds the faulty line, the more points you will receive for this task.

Where to Implement this Task

Implement your solution for *Task 1-2* in **fault_localization.py**.

How to Evaluate this Task

Like in *Task 1-1*, we provide you with tests to evaluate your progress on this task. Note that these tests do not influence the received points for this task. For *Task 1-2*, you have to execute the following:

```
python3.10 -m unittest fault_localization_tests
```

The same for classes and functions as in *Task 1-1* holds for these tests.

Description

For this task, you can use all your knowledge from slicing, statistical debugging, and program analysis to develop your fault localization techniques. You will implement all your strategies in **fault_localization.py**.

We will instrument the code by calling the `Instrumenter.instrument` method from this file and then using the `FaultLocalization` class to verify your results. The `FaultLocalization` needs a `rank(self) -> List[Tuple[str, int]]` method that returns a list of `('function_name', line_number)` that is ordered by the most to the least suspicious location.

We already provide you with an implementation of a collector `EventCollector` that you can leverage. However, you are not required to use it.

We also provide a new library as a playground for you to work with in **lib_fl.py**.

We will evaluate your fault localization by how well it can identify the actual faulty line in a buggy program. The tests in **fault_localization_tests.py** already are an excellent example of this. You can try against these tests, but you can get points for this task even if you cannot pass all of these tests.

Here are some hints and guidelines for you:

- You can reuse any code from the Debugging Book and *Task 1-1*.
- You can leverage any existing technique if you implement it yourself.
- You can implement novel ideas that leverage static or dynamic analysis. You can perform static analysis during the instrumentation and dynamic analysis by dumping and loading events. Your `FaultLocalization` will get your `Instrumenter` so you can extract the static information if you like.
- You are encouraged and should extend your version of `lib_fl.py` where you can track more than dependencies. You can modify and add any method to it but keep in mind that you need to load it to get the information you need in your fault localization.

- You need to adjust the line number this time because we cannot control what you will add during your instrumentation. The easiest way is to leverage the line number while visiting the AST and associate each event you want to collect with its corresponding function name and line number.