DynaPyt:
A Dynamic Analysis Framework for Python

Aryaz Eghbali, Michael Pradel
Software Lab, University of Stuttgart

ESEC/FSE 2022

https://software-lab.org/
Motivation

ML Code

cuda runtime error(2): out of memory

Security:
- Dynamic taint analysis

Performance:
- Memory leak
- Slow operations

Testing:
- Branch coverage
Options

Ad-hoc

- Implement an instrumenter with LibCST
- Nalin has 200+ lines of code for assignment tracking

```python
def leave_SimpleStatementLine(self, node, updated_node):
    body = []
    for node_line in node.body:
        if isinstance(node_line, cst.Assign) or isinstance(node_line, cst.AugAssign):
            line_number = get_metadata(node_line, key=cst.metadata.PositionProvider).start.line

            all_targets = []
            # TODO: We currently only consider assignments/assigns of type a = b and ignore a, b, c = m or b[c] = m.
            if isinstance(node_line, cst.Assign):
                # Go through all targets. Only an Assignment can have multiple targets. Eg: a = b, c = 1, 3, 4
                for assign_target in node_line.targets:
                    target_single_var = assign_target.target.matchers.Name()()
                    if assign_target.matches(assign_target.target_single_var):
                        all_targets.append(assign_target.target.value)
            else:
                if isinstance(node_line, cst.AugAssign):
                    target_single_var = assign_target.target.matchers.Name()()
                    if assign_target.matches(assign_target.target_single_var):
                        all_targets.append(node_line.target.value)
            else:
                return updated_node

        # One call for each target. Eg: a = b = 23. A call each for 'a' & 'b'
        call_expr_nodes = []
        for target_var in all_targets:
            if target_var is None:
                continue
            if target_var is None:
                continue
```
Options

- 70+ lines of code to read the stack properly
- Need low level bytecode and stack operations

```python
sys.settrace

# 70+ lines of code to read the stack properly
# Need low level bytecode and stack operations
```
Options

DynaPyt (this work)

- Just 2 hooks implemented
- At the exact abstraction level of the analysis

```python
from typing import Optional
from .BaseAnalysis import BaseAnalysis

class BranchCoverage(BaseAnalysis):
    def __init__(self):
        self.branches = dict()

    def enter_control_flow(self, dyn_ast: str, lid: int, cond_value: bool) -> Optional[bool]:
        self.branches[(lid, bool(cond_value))] = self.branches.get((lid, bool(cond_value)), 0) + 1

    def end_execution(self):
        for k, v in self.branches.items():
            print(f'Branch {k[0]} taken with condition {k[1]}, \{v\} time$^\prime$ if v == 1 else $^\prime$s$^\prime$')
```
Choose wisely!

<table>
<thead>
<tr>
<th></th>
<th>Engineering Effort</th>
<th>Abstraction Level</th>
<th>Extra Runtime Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad-hoc</td>
<td>High</td>
<td>Matching the analysis</td>
<td>Low</td>
</tr>
<tr>
<td>sys.settrace</td>
<td>Medium</td>
<td>Different from the analysis</td>
<td>High</td>
</tr>
<tr>
<td>DynaPyt</td>
<td>Low</td>
<td>Matching the analysis</td>
<td>Low</td>
</tr>
</tbody>
</table>
DynaPyt

- Instrumentation
- Runtime engine
- Analysis code
Features

- **97 available hooks**
  - Hierarchy: various levels of abstraction
  - Any combination

- **Pay-per-use**
  - Only the used hooks get instrumented
    → overhead only for used hooks

- **Modify execution**
  - Runtime values → in e.g. concolic testing
## Instrument Time

TraceAll analysis: most expensive instrumentation

<table>
<thead>
<tr>
<th>Repository</th>
<th>Instrument time (mm:ss)</th>
<th># of files</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ansible/ansible</td>
<td>6:59</td>
<td>2,188</td>
<td>176,173</td>
</tr>
<tr>
<td>django/django</td>
<td>14:07</td>
<td>3,603</td>
<td>318,602</td>
</tr>
<tr>
<td>keras-team/keras</td>
<td>5:41</td>
<td>678</td>
<td>155,407</td>
</tr>
<tr>
<td>pandas-dev/pandas</td>
<td>12:32</td>
<td>2,727</td>
<td>358,195</td>
</tr>
<tr>
<td>psf/requests</td>
<td>0:16</td>
<td>54</td>
<td>6,370</td>
</tr>
<tr>
<td>Textualize/rich</td>
<td>0:57</td>
<td>178</td>
<td>24,362</td>
</tr>
<tr>
<td>scikit-learn/scikit-learn</td>
<td>6:52</td>
<td>1,419</td>
<td>180,185</td>
</tr>
<tr>
<td>scrapy/scrapy</td>
<td>1:49</td>
<td>505</td>
<td>37,181</td>
</tr>
<tr>
<td>nvbn/thefuck</td>
<td>1:21</td>
<td>620</td>
<td>12,070</td>
</tr>
</tbody>
</table>
Runtime Overhead

DynaPyt Analyses

Python sys.settrace (opcode)

Overhead factor

ansible | django | keras | pandas | requests | rich | scikit-learn | scrapy | thefuck
### Usage

**Install:**

```bash
pip install dynapyt
```

**Instrument code:**

```bash
python -m dynapyt.run_instrumentation --directory <dir> --analysis MLMemoryAnalysis
```

**Run analysis:**

```bash
python -m dynapyt.run_analysis --entry <main.py> --analysis MLMemoryAnalysis
```
## Analysis Simplicity

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Analysis hooks</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>BranchCoverage</td>
<td>Measures how often each branch gets covered</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>CallGraph</td>
<td>Computes a dynamic call graph</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>KeyInList</td>
<td>Warns about performance anti-pattern of linearly search through a list</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>MLMemory</td>
<td>Warns about memory leak issues in deep learning code</td>
<td>4</td>
<td>29</td>
</tr>
<tr>
<td>SimpleTaint</td>
<td>Taint analysis useful to, e.g., detect SQL injections</td>
<td>7</td>
<td>53</td>
</tr>
<tr>
<td>AllEvents</td>
<td>Implements the runtime_event analysis hook to trace all events</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
Future Work

For DynaPyt:

- Write to attributes, as a multi-write, to a tuple
- async/await

With DynaPyt:

- Early detection of ML issues
- Creating datasets of Python executions
Q&A

DynaPyt: Dynamic Analysis Framework for Python

★ Ease of analysis implementation
★ Low overhead runtime

Install:
  pip install dynapyt

Code & documentation:
  https://github.com/sola-st/DynaPyt
Execution Faithfulness

- Preserve original execution
  - All above 97.8% passing tests
  - Part of the difference is due to execution stack accesses