DynaPyt:

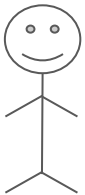
A Dynamic Analysis Framework for Python

Aryaz Eghbali, Michael Pradel
Software Lab, University of Stuttgart

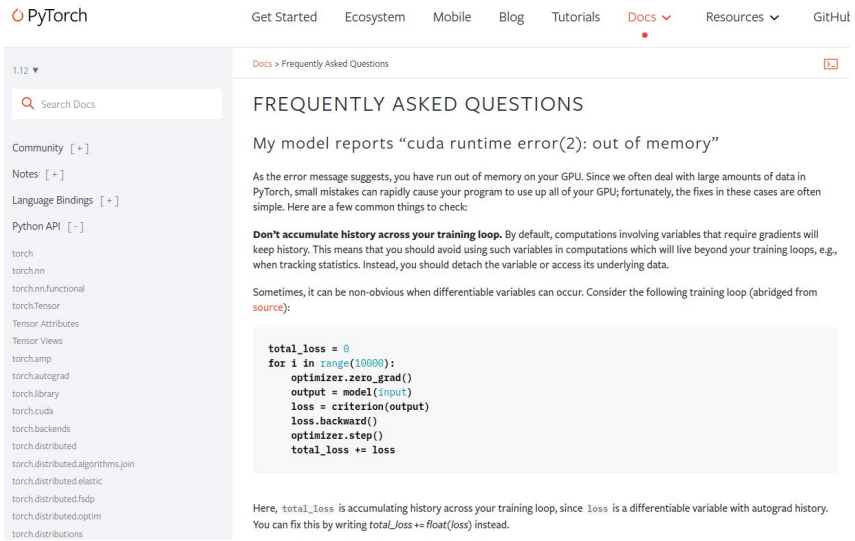
ESEC/FSE 2022

https://software-lab.org/

Motivation



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

```
55      def leave_SimpleStatementLine(self, node, updated_node):
56          body = []
57          for node_line in node.body:
58              if isinstance(node_line, cst.Assign) or isinstance(node_line, cst.AugAssign):
59                  line_number = str(self.get_metadata(node=node, key=cst.metadata.PositionProvider).start.line)
60                  all_targets = []
61                  # TODO: We currently only consider assignments/augassigns of type a = b and ignore a.b.c = m or b[c] = m.
62                  if isinstance(node_line, cst.Assign):
63                      # Go through all targets. Only an Assignment can have multiple targets. Eg. a=b=23 or a,b,c = 2,3,4
64                      for assign_target in node_line.targets:
65                          target_single_var = matchers.AssignTarget(target=matchers.Name())
66                          if matchers.matches(assign_target, target_single_var):
67                              all_targets.append(assign_target.target.value)
68                          # Track values of type a,b,c = 1,2,3
69                          target_tuple = matchers.AssignTarget(target=matchers.Tuple())
70                          if matchers.matches(assign_target, target_tuple):
71                              for elem in assign_target.children:
72                                  for v in elem.children:
73                                      if matchers.matches(v, matchers.Element(value=matchers.Name())):
74                                          all_targets.append(v.value.value)
75
76                      if not len(all_targets):
77                          return updated_node
78                  elif isinstance(node_line, cst.AugAssign):
79                      target_single_var = matchers.AugAssign(target=matchers.Name())
80                      if matchers.matches(node_line, target_single_var):
81                          all_targets.append(node_line.target.value)
82                      else:
83                          return updated_node
84                  # One call for each target. Eg a = b = 23. A call each for 'a' & 'b'
85                  call_expr_nodes = []
86                  for target_var in all_targets:
87                      # dc = cst.Dict([
88                      #     cst.DictElement(cst.Name('variable_name'), cst.Name(target_var)),
89                      #     cst.DictElement(cst.Name('line_number'), cst.Integer(line_number)),
90                      #     cst.DictElement(cst.Name('value'), node_line.value)
91                      # ]
```

Patra, Jibesh, and Michael Pradel. "Nalin: learning from runtime behavior to find name-value inconsistencies in jupyter notebooks." *Proceedings of the 44th International Conference on Software Engineering*. 2022.

Options

sys.settrace

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

```python
PTR_SIZE = sizeof(POINTER(py_object))
F_VALUESTACK_OFFSET = sizeof(Frame) - 2 * PTR_SIZE
F_STACKTOP_OFFSET = sizeof(Frame) - PTR_SIZE


class OpStack(Sequence[Any]):
    __slots__ = ("_frame", "_len")

    def __init__(self, frame):
        self._frame = Frame.from_address(id(frame))
        stack_start_addr = c_ssize_t.from_address(id(frame) + F_VALUESTACK_OFFSET).value
        stack_top_addr = c_ssize_t.from_address(id(frame) + F_STACKTOP_OFFSET).value
        self._len = (stack_top_addr - stack_start_addr) // PTR_SIZE

    def __repr__(self) -> str:
        if not self:
            return "<OpStack> (empty)>"
        return "<OpStack ({})>\n- {}\n".format(
            len(self),
            "\n- ".join(repr(o) for o in reversed(self)),
        )

    def __len__(self):
        return self._len

    def _preproc_slice(self, idx: Optional[int], default: int) -> int:
        if idx is None:
            return default
        if idx < -self._len or idx >= self._len:
            raise IndexError(idx)
        if idx < 0:
            return idx + self._len
        return idx

    def __getitem__(self, item: Union[int, slice]) -> Any:
        if isinstance(item, int):
            if item < -self._len or item >= self._len:
                raise IndexError(item)
            if item < 0:
                return self._frame.f_stacktop[item]
            return self._frame.f_valuestack[item]

        if isinstance(item, slice):
            item = slice(
                self._preproc_slice(item.start, 0),
                self._preproc_slice(item.stop, self._len),
                item.step
            )
            return self._frame.f_valuestack[item]

        raise TypeError(item)
```

```python
import sys
import opcode
from get_stack import OpStack

branches = dict()

def show_trace(frame, event, arg):
    global branches
    frame.f_trace_opcodes = True
    code = frame.f_code
    offset = frame.f_lasti

    if opcode.opname[code.co_code[offset]] in ['POP_JUMP_IF_FALSE', 'POP_JUMP_IF_TRUE']:
        st = OpStack(frame)
        branches[(frame.f_lineno, st[0])] = branches.get((frame.f_lineno, st[0]), 0) + 1
    return show_trace

sys.settrace(show_trace)
```

Options

DynaPyt (this work)

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

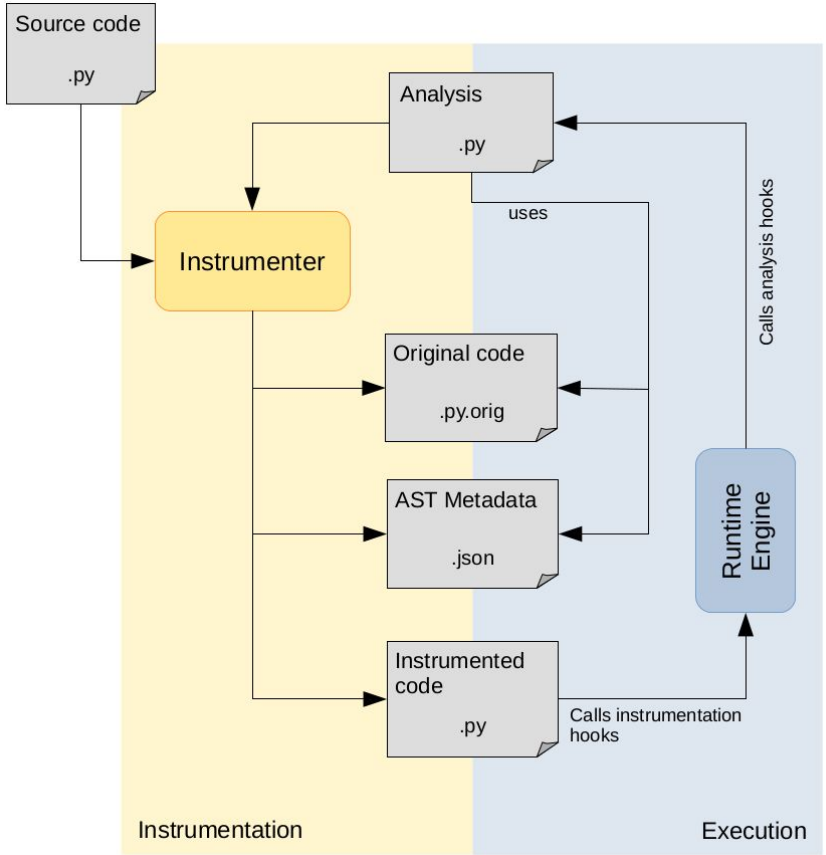
```python
from typing import Optional
from .BaseAnalysis import BaseAnalysis

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

    def enter_control_flow(self, dyn_ast: str, iid: int, cond_value: bool) -> Optional[bool]:
        self.branches[(iid, bool(cond_value))] = self.branches.get((iid, 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{"" if v == 1 else "s"}')
```

Choose wisely!

	Engineering Effort	Abstraction Level	Extra Runtime Overhead
Ad-hoc	High	Matching the analysis	Low
sys.settrace	Medium	Different from the analysis	High
DynaPyt	Low	Matching the analysis	Low

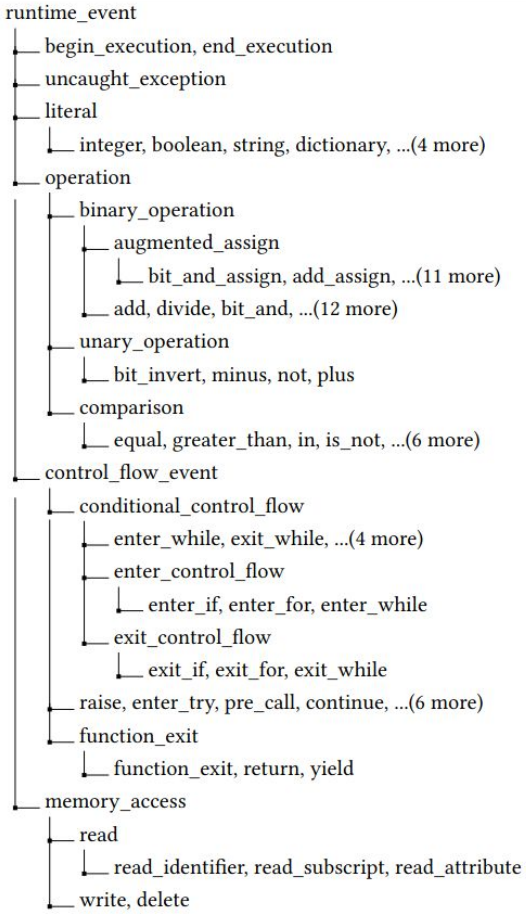
DynaPyt

- 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

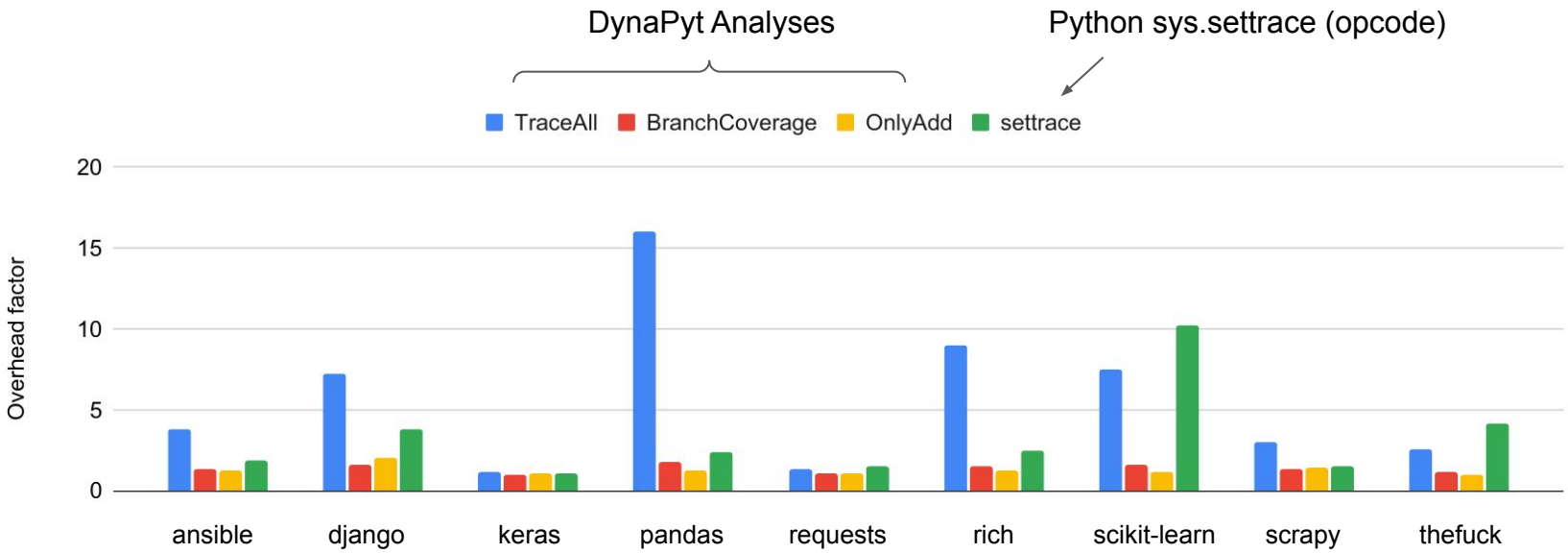


Instrument Time

TraceAll analysis: most expensive instrumentation

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

Runtime Overhead



Usage

Install:

```
pip install dynapyt
```

Instrument code:

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

Run analysis:

```
python -m dynapyt.run_analysis --entry <main.py> --analysis MLMemoryAnalysis
```

```python
# Inspired by https://pytorch.org/docs/stable/notes/faq.html#my-model-reports-cuda-runtime-error-2-out-of-memory
from collections import defaultdict
from .BaseAnalysis import BaseAnalysis

class MLMemoryAnalysis(BaseAnalysis):
    def __init__(self) -> None:
        super().__init__()
        self.in_ctrl_flow = []
        self.threshold = 3
        self.memory_leak = defaultdict(lambda: 0)
        self.last_opr = None

    def enter_control_flow(self, dyn_ast, iid, condition):
        self.last_opr = None
        if (len(self.in_ctrl_flow) > 0) and (self.in_ctrl_flow[-1] != iid):
            self.in_ctrl_flow.append(iid)

    def exit_control_flow(self, dyn_ast, iid):
        self.last_opr = None
        self.in_ctrl_flow.pop()

    def binary_operation(self, dyn_ast, iid, opr, left, right, res):
        if (len(self.in_ctrl_flow) > 0) and right.requires_grad:
            self.last_opr = iid
        else:
            self.last_opr = None

    def write(self, dyn_ast, iid, left, right):
        if (len(self.in_ctrl_flow) > 0) and right.requires_grad and (self.last_opr is not None):
            cur = (iid, self.in_ctrl_flow[-1])
            self.memory_leak[cur] += 1
            if self.memory_leak[cur] > 3:
                print('Memory issue detected')
                exit(1)
        self.last_opr = None
```

Analysis Simplicity

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

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

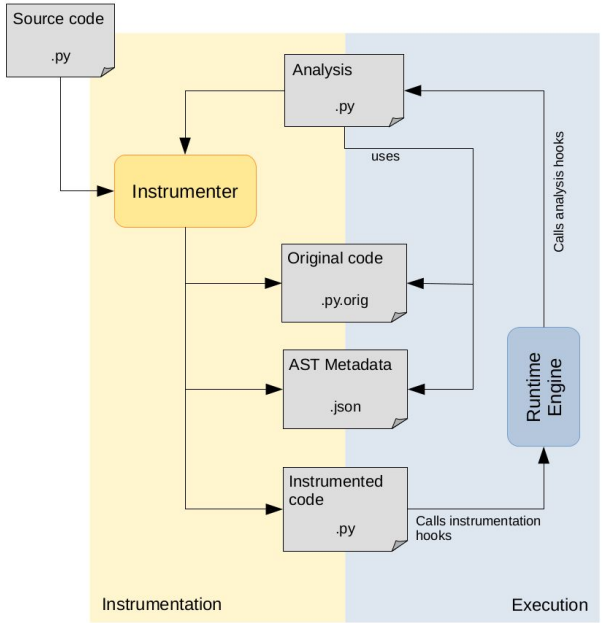
- ★ Ease of analysis implementation
- ★ Low overhead runtime

Install:

```
pip install dynapyt
```

Code & documentation:

```
https://github.com/sola-st/DynaPyt
```



- runtime_event
  - begin_execution, end_execution
  - uncaught_exception
  - literal
    - integer, boolean, string, dictionary, ...(4 more)
  - operation
    - binary_operation
      - augmented_assign
        - bit_and_assign, add_assign, ...(11 more)
      - add, divide, bit_and, ...(12 more)
    - unary_operation
      - bit_invert, minus, not, plus
    - comparison
      - equal, greater_than, in, is_not, ...(6 more)
  - control_flow_event
    - conditional_control_flow
      - enter_while, exit_while, ...(4 more)
      - enter_control_flow
        - enter_if, enter_for, enter_while
      - exit_control_flow
        - exit_if, exit_for, exit_while
    - raise, enter_try, pre_call, continue, ...(6 more)
    - function_exit
      - function_exit, return, yield
  - memory_access
    - read
      - read_identifier, read_subscript, read_attribute
    - write, delete

Execution Faithfulness

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