DynaPyt:
A Dynamic Analysis Framework for Python

Michael Pradel
Software Lab – University of Stuttgart
Joint work with Aryaz Eghbali
Python:

- Extremely popular
- Highly dynamic language
- Underrepresented as a target language in research
Dynamic Analysis for Python

Python:
- Extremely popular
- Highly dynamic language
- Underrepresented as a target language in research
Dynamic Analysis for Python

Python:

- Extremely popular
- Highly dynamic language
- Underrepresented as a target language in research
Dynamic Analysis for Python

Python:

- Extremely popular
- Highly dynamic language
- Underrepresented as a target language in research

Perfect target for dynamic analyses!
Implementing a Dynamic Analysis

■ Option 1: Implement from scratch
  □ Custom source-level instrumentation
  □ Custom bytecode-level instrumentation

■ Option 2: Built-in constructs
  □ `sys.settrace`: Observe every line or opcode
Implementing a Dynamic Analysis

- **Option 1:** Implement from scratch
  - Custom source-level instrumentation
  - Custom bytecode-level instrumentation

- **Option 2:** Built-in constructs
  - `sys.settrace`: Observe every line or opcode

High engineering effort, repeated for each analysis
Implementing a Dynamic Analysis

- **Option 1:** Implement from scratch
  - Custom source-level instrumentation
  - Custom bytecode-level instrumentation

- **Option 2:** Built-in constructs
  - `sys.settrace`: Observe every line or opcode

Abstraction mismatch, observation-only, relatively high overhead
## Dynamic Analysis Frameworks

<table>
<thead>
<tr>
<th>Target language</th>
<th>Analysis framework(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>Jalangi, NodeProf</td>
</tr>
<tr>
<td>WebAssembly</td>
<td>Wasabi</td>
</tr>
<tr>
<td>Java</td>
<td>DiSL, RoadRunner</td>
</tr>
<tr>
<td>x86 binaries</td>
<td>Pin, Valgrind</td>
</tr>
<tr>
<td>Python</td>
<td>???</td>
</tr>
</tbody>
</table>
This Talk: DynaPyt

First **general-purpose dynamic analysis framework** for Python

- Hierarchy of runtime events
- Pay-per-use principle
- Observe and modify all runtime behavior
- Six client analyses (and more coming)
Overview of DynaPyt

Source code (.py)

Instrumententer

Analysis (.py)

Instrumented code (.py)

Runtime engine

calls
Overview of DynaPyt

Source code (.py)

Instrumenter

Analysis (.py)

Original code (.py.orig)

AST metadata (.json)

Instrumented code (.py)

Runtime engine

calls

uses

calls
Example 1: Branch Coverage

```python
from collections import defaultdict
from .BaseAnalysis import BaseAnalysis

class BranchCoverage(BaseAnalysis):
    def __init__(self):
        self.branches = defaultdict(lambda: 0)

    def enter_control_flow(self, ast, iid, condition):
        self.branches[(iid, condition)] += 1
```
Example 1: Branch Coverage

Build upon base analysis

```python
from collections import defaultdict
from .BaseAnalysis import BaseAnalysis

class BranchCoverage(BaseAnalysis):
    def __init__(self):
        self.branches = defaultdict(lambda: 0)

    def enter_control_flow(self, ast, iid, condition):
        self.branches[(iid, condition)] += 1
```
Example 1: Branch Coverage

Build upon base analysis

```python
from collections import defaultdict
from .BaseAnalysis import BaseAnalysis

class BranchCoverage(BaseAnalysis):
    def __init__(self):
        self.branches = defaultdict(lambda: 0)

    def enter_control_flow(self, ast, iid, condition):
        self.branches[(iid, condition)] += 1
```

Register for all control flow events
Example 1: Branch Coverage

Build upon base analysis

```python
from collections import defaultdict
from .BaseAnalysis import BaseAnalysis

class BranchCoverage(BaseAnalysis):
    def __init__(self):
        self.branches = defaultdict(lambda: 0)

    def enter_control_flow(self, ast, iid, condition):
        self.branches[(iid, condition)] += 1
```

Register for all control flow events

Initialize and update branch counts
Example 2: Key-in-List Anti-Pattern

Performance anti-pattern:

```python
# d is the list of words read from a large file
# queries is a list of words to check
for query in queries:
    if query in d:
        print(f'Found {query}')
```

Example 2: Key-in-List Anti-Pattern

Performance anti-pattern:

```python
# d is the list of words read from a large file
# queries is a list of words to check
for query in queries:
    if query in d:
        print(f'Found {query}')
```

Slow, because repeatedly iterates through the list
Example 2: Key-in-List Anti-Pattern

Analysis to find instances of this pattern:

```python
from .BaseAnalysis import BaseAnalysis

class KeyInListAnalysis(BaseAnalysis):
    def __init__(self):
        self.threshold = 100

    def _in(self, ast, iid, left, right, result):
        if (isinstance(right, list) and
            len(right) > self.threshold):
            print('Performance warning')
```

```python
from .BaseAnalysis import BaseAnalysis
class KeyInListAnalysis(BaseAnalysis):
def __init__(self):
def _in(self, ast, iid, left, right, result):
    if (isinstance(right, list) and
        len(right) > self.threshold):
        print('Performance warning')
```
Example 2: Key-in-List Anti-Pattern

Analysis to find instances of this pattern:

```python
from .BaseAnalysis import BaseAnalysis

class KeyInListAnalysis(BaseAnalysis):
    def __init__(self):
        self.threshold = 100

    def _in(self, ast, iid, left, right, result):
        if (isinstance(right, list) and len(right) > self.threshold):
            print('Performance warning ')

Register for binary operator in
```
Example 2: Key-in-List Anti-Pattern

Analysis to find instances of this pattern:

```python
from .BaseAnalysis import BaseAnalysis

class KeyInListAnalysis(BaseAnalysis):
    def __init__(self):
        self.threshold = 100

    def __in__(self, ast, iid, left, right, result):
        if (isinstance(right, list) and
            len(right) > self.threshold):
            print('Performance warning')

Register for binary operator `in`

Warn when used on long lists
Event Hierarchy

- Many **different runtime events** (97)
- Instead of hard-coding an event granularity:
  **Hierarchy of event APIs** to register for
Event Hierarchy

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
Event Hierarchy

- runtime_event
  - 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
Source-to-Source Instrumentation

- AST-based transformation rules
- Modify expressions and statements to inject calls into the runtime engine
Examples (1)

Evaluating an integer literal:

23

_int_ (f, iid, 23)

f, iid, and opid are placeholders for filename, instruction id, and operator id
Examples (1)

Evaluating an integer literal:

23

_notify_ (f, iid, 23)

Notify runtime engine about the literal

f, iid, and opid are placeholders for filename, instruction id, and operator id
Examples (2)

For-in loops:

```python
for x in coll:
    # stmts

for x in _gen_(f, iid, coll):
    # stmts

else:
    _exit_for_(f, iid)
```

*f, iid, and opid are placeholders for filename, instruction id, and operator id*
Examples (2)

For-in loops:

```
for x in coll:
    # stmts

for x in _gen_(f, iid, coll):
    # stmts

else:
    _exit_for_(f, iid)
```

- Indicate that generator expression produces another value
- Indicate that loop has terminated

*f, iid, and opid are placeholders for filename, instruction id, and operator id*
Examples (3)

Complex expression and assignment:

\[
c = a + b
\]

\[
c = \_write\_(f, 
    iid, \_binary\_op\_(f, iid, 
        lambda: a, opid, lambda: b), [lambda: c])
\]

\[f, iid, and opid are placeholders for filename, instruction id, and operator id\]
Examples (3)

Complex expression and assignment:

\[ c = a + b \]

- Wrap subexpressions into a lambda functions to delay evaluation
- Runtime engine controls when to evaluate each expression
- Analysis may change values

\[ c = \text{\_write\_}(f, iid, \text{\_binary\_op\_}(f, iid, \text{\_lambda\_}: a, opid, \text{\_lambda\_}: b), [\text{\_lambda\_}: c]) \]

\( f, iid, \text{ and } opid \) are placeholders for filename, instruction id, and operator id
Examples (3)

Complex expression and assignment:

\[ c = a + b \]

Analysis interested in writes can see old and new value

\[ c = \text{\_write\_}(f, \ \text{\_binary\_op\_}(f, \ iid, \ \lambda: a, \ opid, \ \lambda: b), \ [\lambda: c]) \]

\( f, \ iid, \) and \( opid \) are placeholders for filename, instruction id, and operator id
Pay-per-Use Principle

- Selective instrumentation
- Inject only those calls needed for the analysis
Evaluation

- **Benchmarks**
  - 9 popular open-source projects
  - 1.3 MLoC, 153k test cases

- **Research questions**
  - Efficiency of instrumentation
  - Faithfulness to original semantics
  - Complexity of client analyses
  - Runtime overhead
## Efficiency of Instrumentation

<table>
<thead>
<tr>
<th>Repository</th>
<th>Instrument time (mm:ss)</th>
<th>Python files</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ansible/ansible</td>
<td>06:59</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>05:41</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>00:16</td>
<td>54</td>
<td>6,370</td>
</tr>
<tr>
<td>Textualize/rich</td>
<td>00:57</td>
<td>178</td>
<td>24,362</td>
</tr>
<tr>
<td>scikit-learn/scikit-learn</td>
<td>06:52</td>
<td>1,419</td>
<td>180,185</td>
</tr>
<tr>
<td>scrapy/scrapy</td>
<td>01:49</td>
<td>505</td>
<td>37,181</td>
</tr>
<tr>
<td>nvbn/thefuck</td>
<td>01:21</td>
<td>620</td>
<td>12,070</td>
</tr>
</tbody>
</table>
## Efficiency of Instrumentation

<table>
<thead>
<tr>
<th>Repository</th>
<th>Instrument time (mm:ss)</th>
<th>Python files</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ansible/ansible</td>
<td>06:59</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>05:41</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>00:16</td>
<td>54</td>
<td>6,370</td>
</tr>
<tr>
<td>Textualize/rich</td>
<td>00:57</td>
<td>178</td>
<td>24,362</td>
</tr>
<tr>
<td>scikit-learn/scikit-learn</td>
<td>06:52</td>
<td>1,419</td>
<td>180,185</td>
</tr>
<tr>
<td>scrapy/scrapy</td>
<td>01:49</td>
<td>505</td>
<td>37,181</td>
</tr>
<tr>
<td>nvbn/thefuck</td>
<td>01:21</td>
<td>620</td>
<td>12,070</td>
</tr>
</tbody>
</table>

2.4 seconds per 1,000 LoC
Faithfulness to Original Semantics

Passing test cases:

<table>
<thead>
<tr>
<th># without instrum.</th>
<th>% after instrum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,651</td>
<td>93.4%</td>
</tr>
<tr>
<td>189</td>
<td>98.4%</td>
</tr>
<tr>
<td>402</td>
<td>99.8%</td>
</tr>
<tr>
<td>136,898</td>
<td>99.8%</td>
</tr>
<tr>
<td>39</td>
<td>100.0%</td>
</tr>
<tr>
<td>568</td>
<td>99.5%</td>
</tr>
<tr>
<td>9,400</td>
<td>97.8%</td>
</tr>
<tr>
<td>1,841</td>
<td>99.6%</td>
</tr>
<tr>
<td>1,798</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
Faithfulness to Original Semantics

Passing test cases:

<table>
<thead>
<tr>
<th># without instrum.</th>
<th>% after instrum.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,651</td>
<td>93.4%</td>
</tr>
<tr>
<td>189</td>
<td>98.4%</td>
</tr>
<tr>
<td>402</td>
<td>99.8%</td>
</tr>
<tr>
<td>136,898</td>
<td>99.8%</td>
</tr>
<tr>
<td>39</td>
<td>100.0%</td>
</tr>
<tr>
<td>568</td>
<td>99.5%</td>
</tr>
<tr>
<td>9,400</td>
<td>97.8%</td>
</tr>
<tr>
<td>1,841</td>
<td>99.6%</td>
</tr>
<tr>
<td>1,798</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

Reasons why not yet 100%

- Assertions that inspect the stack
- Two known and to-be-fixed bugs in the instrumenter
# Example Analyses

<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>
Runtime Overhead

The graph shows the runtime overhead for different projects labeled from 1 to 9. The x-axis represents the project number, while the y-axis represents the overhead factor. The graph includes four categories: TraceAll, BranchCoverage, OnlyAdd, and settrace. The bars indicate the overhead for each category in each project.
Runtime Overhead

Trace all events: Most expensive analysis

![Graph showing overhead factor for different projects and trace options: TraceAll, BranchCoverage, OnlyAdd, settrace. The graph illustrates the overhead factor across various projects.](image-url)
Runtime Overhead

All control flow branching points

- TraceAll
- BranchCoverage
- OnlyAdd
- settrace

Overhead factor vs Project #
Runtime Overhead

All “plus” operations

![Graph showing runtime overhead for different projects with various operation counts.]
Runtime Overhead

DynaPyt is 6%–87% faster for lightweight analyses
Conclusions

- **DynaPyt**: First dynamic analysis framework for Python
  - Event hierarchy
  - Pay-per-use principle

- **More details**:
  - Upcoming FSE’22 paper
  - [https://github.com/sola-st/DynaPyt](https://github.com/sola-st/DynaPyt)

**Talk to me about analysis ideas!**