DynaPyt: A Dynamic Analysis Framework for Python

Michael Pradel

Software Lab – University of Stuttgart

Joint work with Aryaz Eghbali

SOLA SoftwareLab University of Stuttgart Germany

European Research Council

Dynamic Analysis for Python

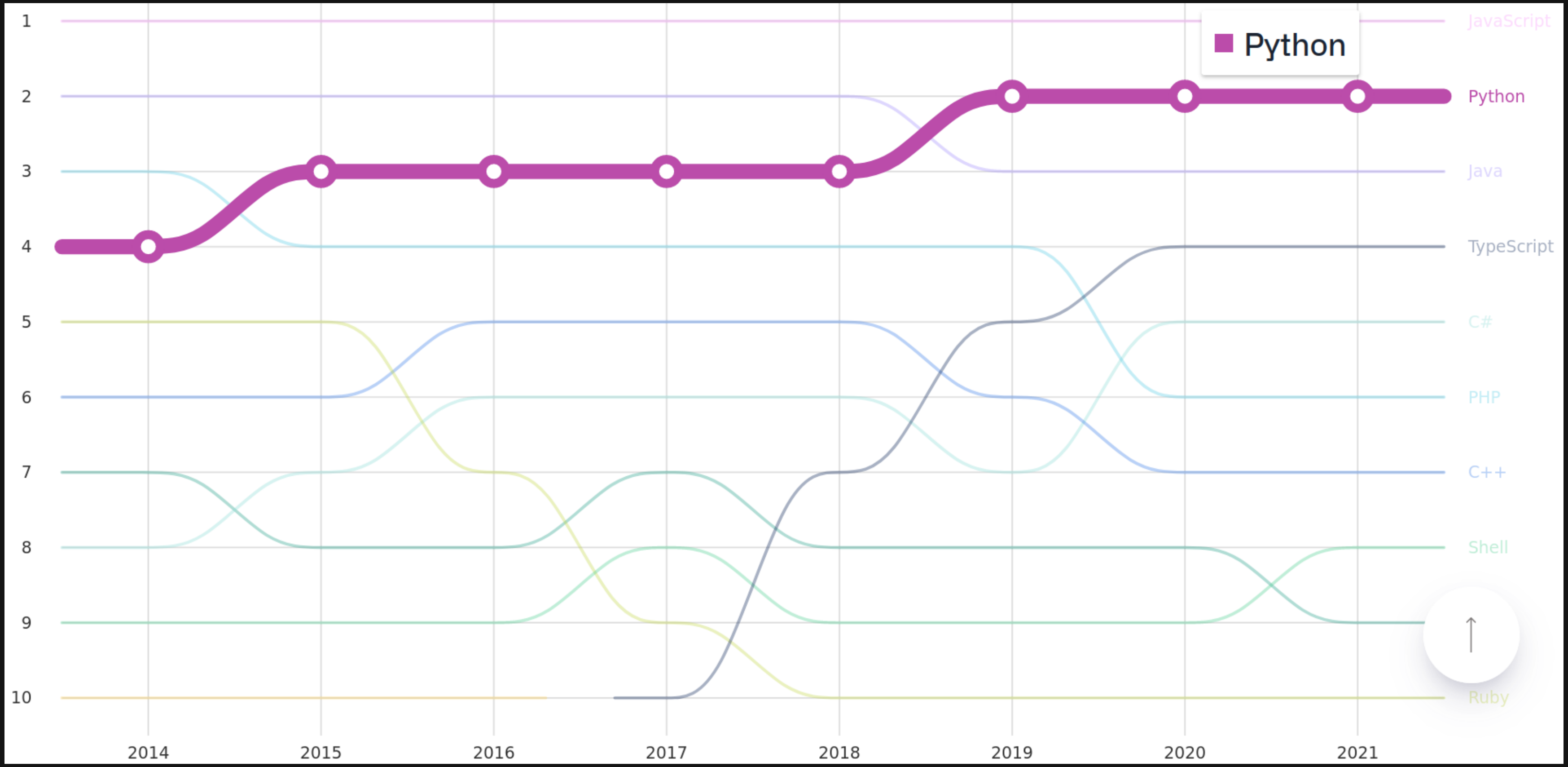
Python:

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

Dynamic Analysis for Python

Python:

- Extremely **popular**



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

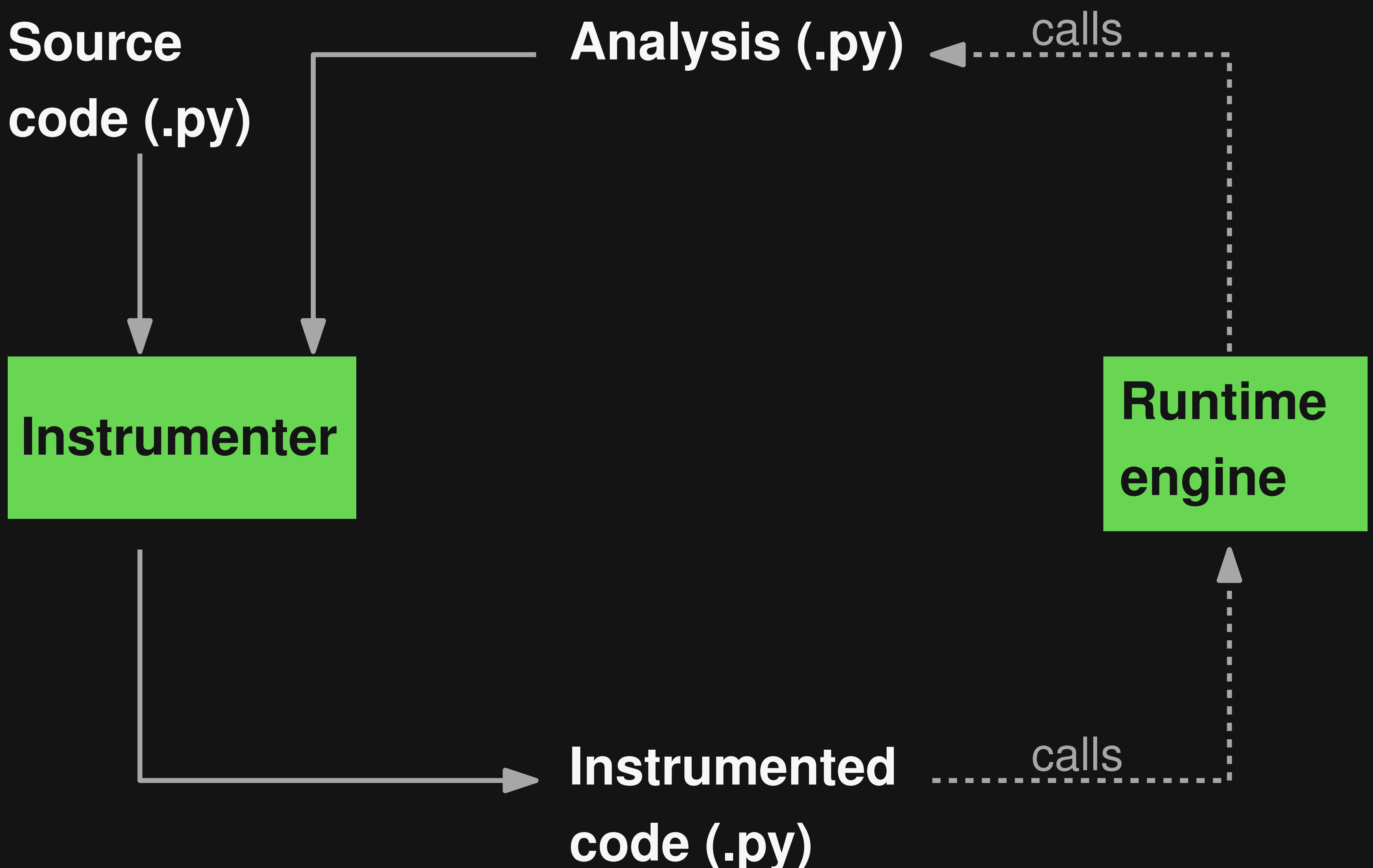
Target language	Analysis framework(s)
JavaScript	Jalangi, NodeProf
WebAssembly	Wasabi
Java	DiSL, RoadRunner
x86 binaries	Pin, Valgrind
Python	???

This Talk: DynaPyt

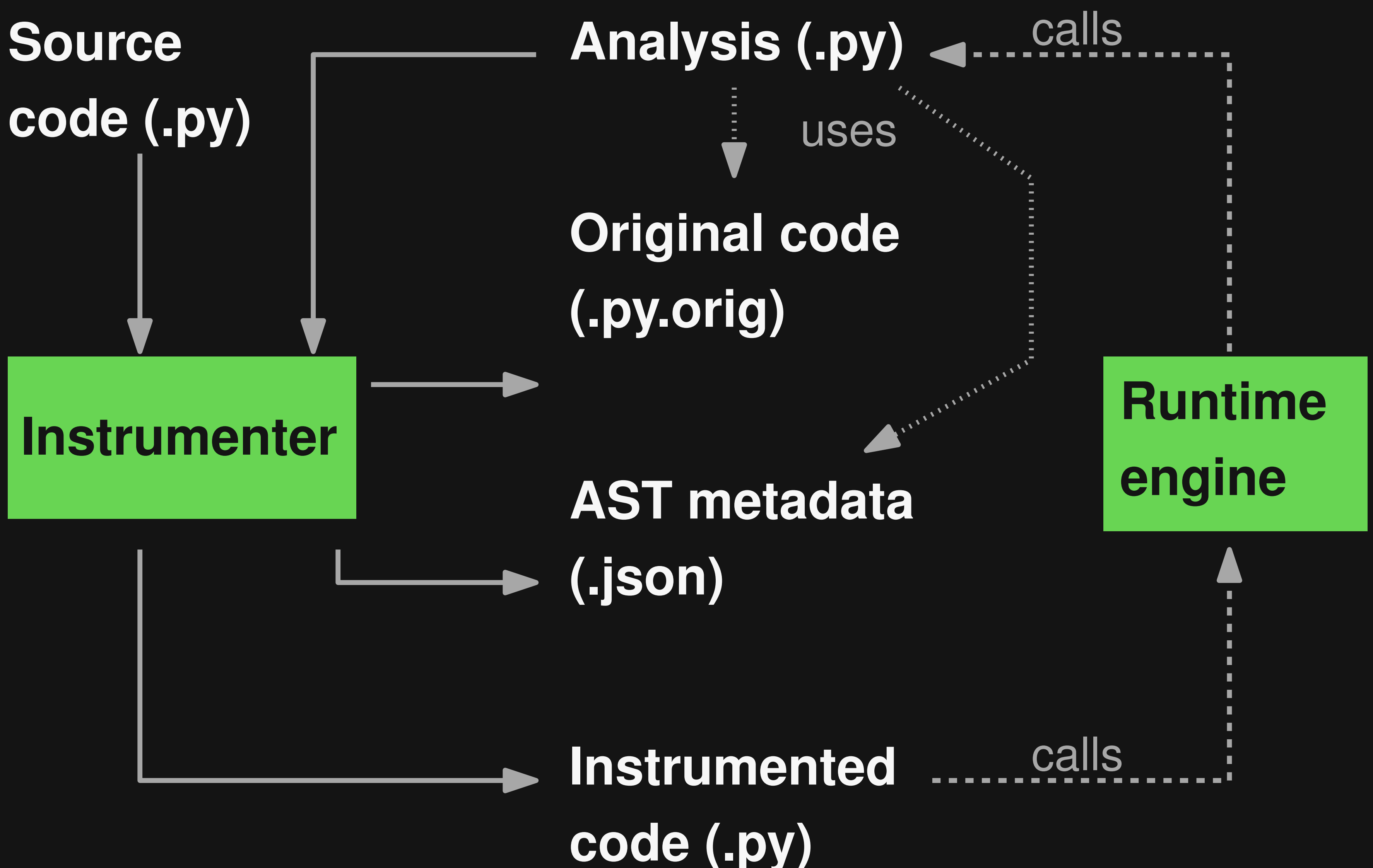
First general-purpose dynamic analysis framework for Python

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

Overview of DynaPyt



Overview of DynaPyt



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

```
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

```
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

```
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:

```
# 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:

```
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')
```

Example 2: Key-in-List Anti-Pattern

Analysis to find instances of this pattern:

```
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:

```
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
```

↓

```
_int_(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:

```
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
```

```
c = _write_(f,
     iid, _binary_op_(f, iid,
          lambda: a, opid, lambda: b), [lambda: c])
```

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

`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])
```

Analysis interested in writes can see old and new value

`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

Repository	Instrument time (mm:ss)	Python files	Lines of code
ansible/ansible	06:59	2,188	176,173
django/django	14:07	3,603	318,602
keras-team/keras	05:41	678	155,407
pandas-dev/pandas	12:32	2,727	358,195
psf/requests	00:16	54	6,370
Textualize/rich	00:57	178	24,362
scikit-learn/scikit-learn	06:52	1,419	180,185
scrapy/scrapy	01:49	505	37,181
nvbn/thefuck	01:21	620	12,070

Efficiency of Instrumentation

Repository	Instrument time (mm:ss)	Python files	Lines of code
ansible/ansible	06:59	2,188	176,173
django/django	14:07	3,603	318,602
keras-team/keras	05:41	678	155,407
pandas-dev/pandas	12:32	2,727	358,195
psf/requests	00:16	54	6,370
Textualize/rich	00:57	178	24,362
scikit-learn/scikit-learn	06:52	1,419	180,185
scrapy/scrapy	01:49	505	37,181
nvbn/thefuck	01:21	620	12,070

2.4 seconds per 1,000 LoC

Faithfulness to Original Semantics

Passing test cases:

# without instrum.	% after instrum.
1,651	93.4%
189	98.4%
402	99.8%
136,898	99.8%
39	100.0%
568	99.5%
9,400	97.8%
1,841	99.6%
1,798	100.0%

Faithfulness to Original Semantics

Passing test cases:

# without instrum.	% after instrum.
1,651	93.4%
189	98.4%
402	99.8%
136,898	99.8%
39	100.0%
568	99.5%
9,400	97.8%
1,841	99.6%
1,798	100.0%

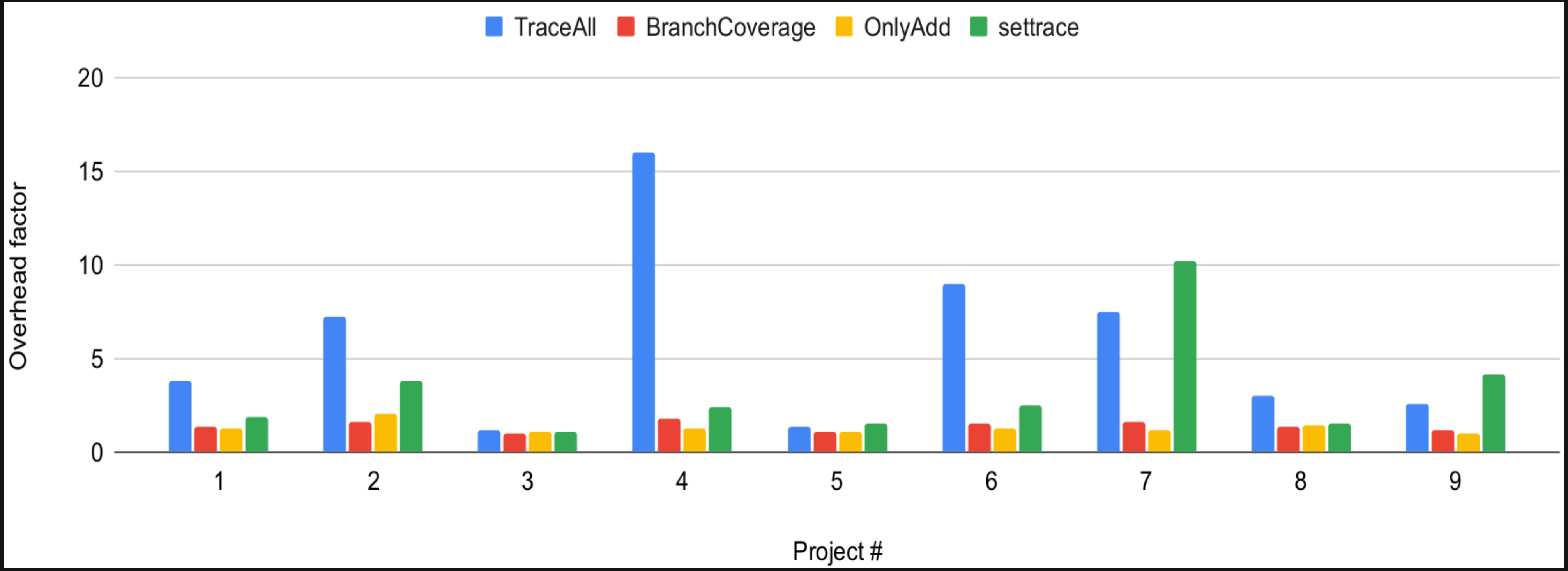
Reasons why not yet 100%

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

Example Analyses

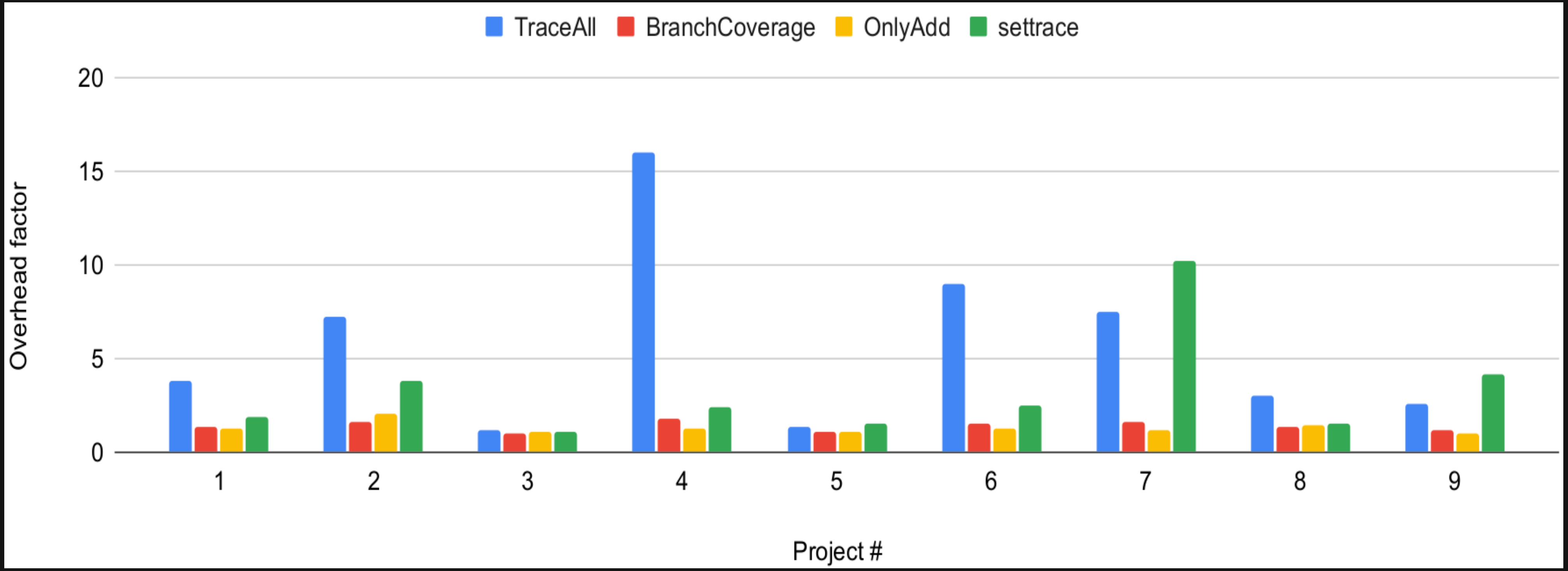
Name	Description	Analysis hooks	LoC
BranchCoverage	Measures how often each branch gets covered	1	6
CallGraph	Computes a dynamic call graph	1	19
KeyInList	Warns about performance anti-pattern of linearly search through a list	2	10
MLMemory	Warns about memory leak issues in deep learning code	4	29
SimpleTaint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

Runtime Overhead



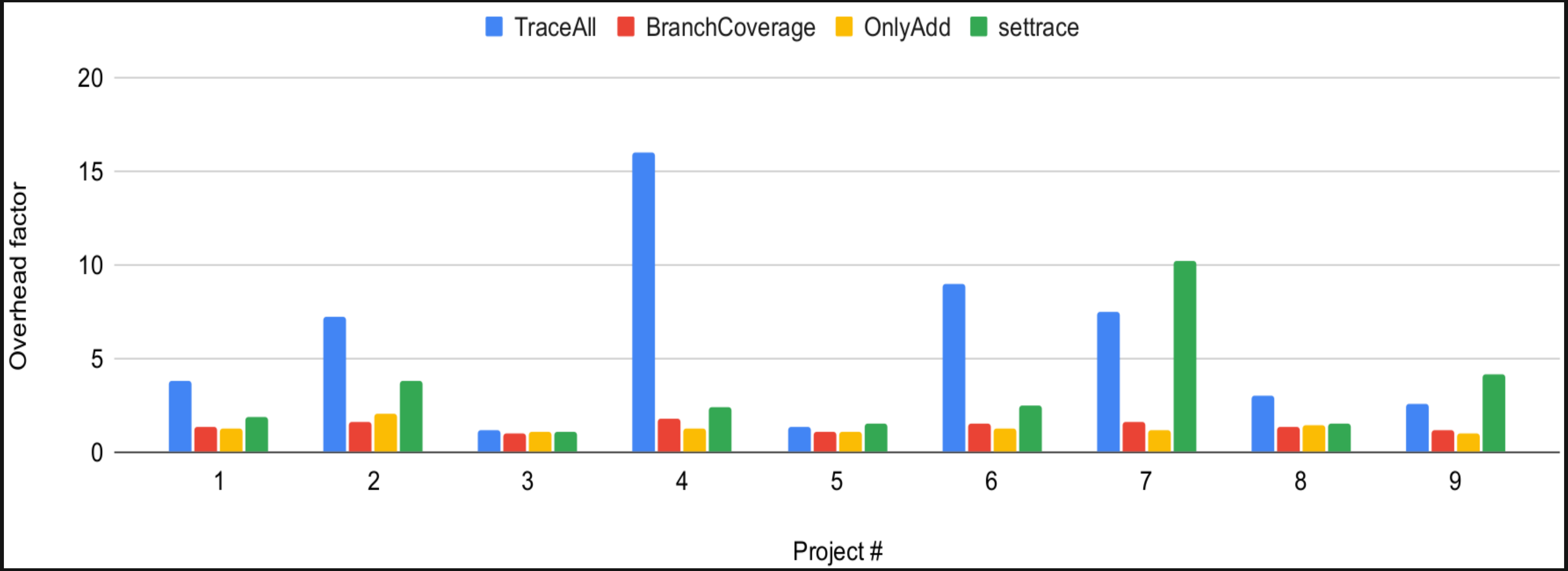
Runtime Overhead

Trace all events: Most expensive analysis

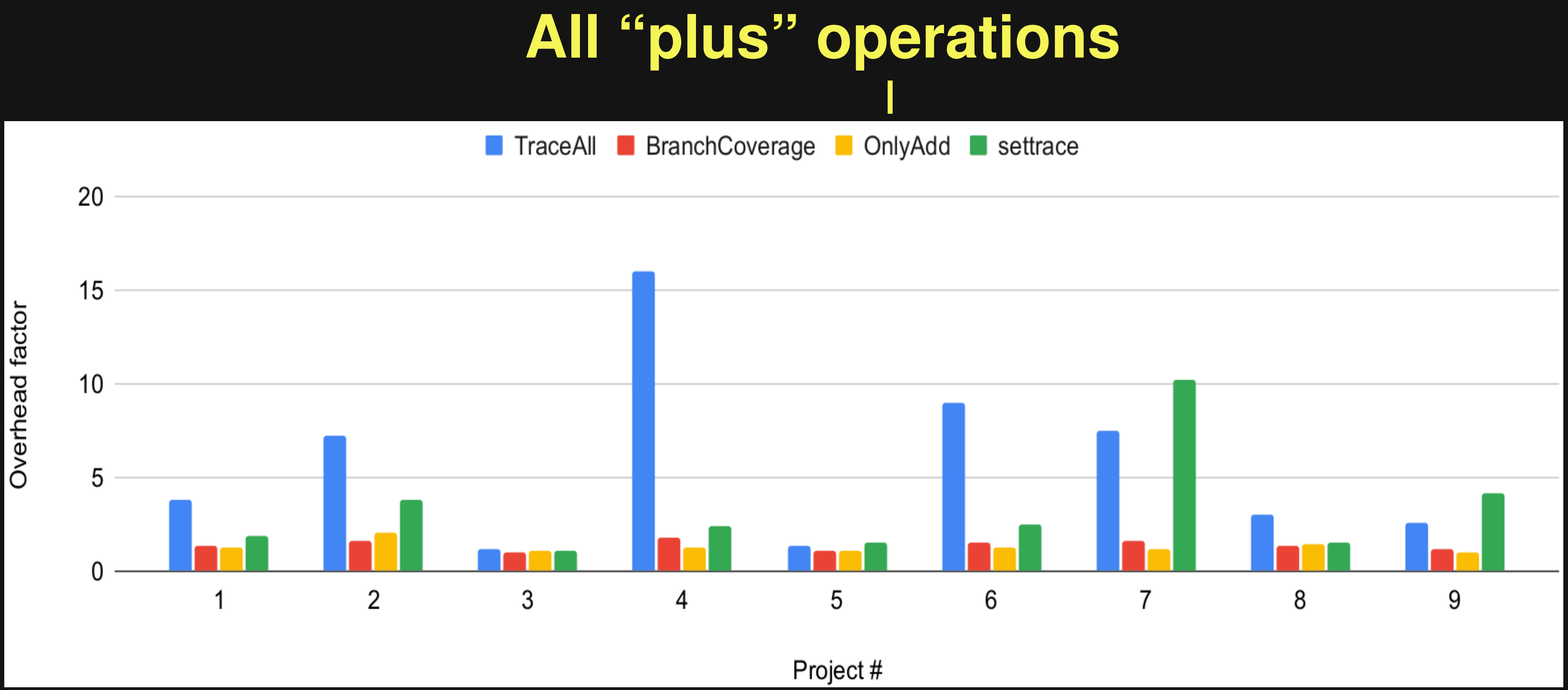


Runtime Overhead

All control flow branching points

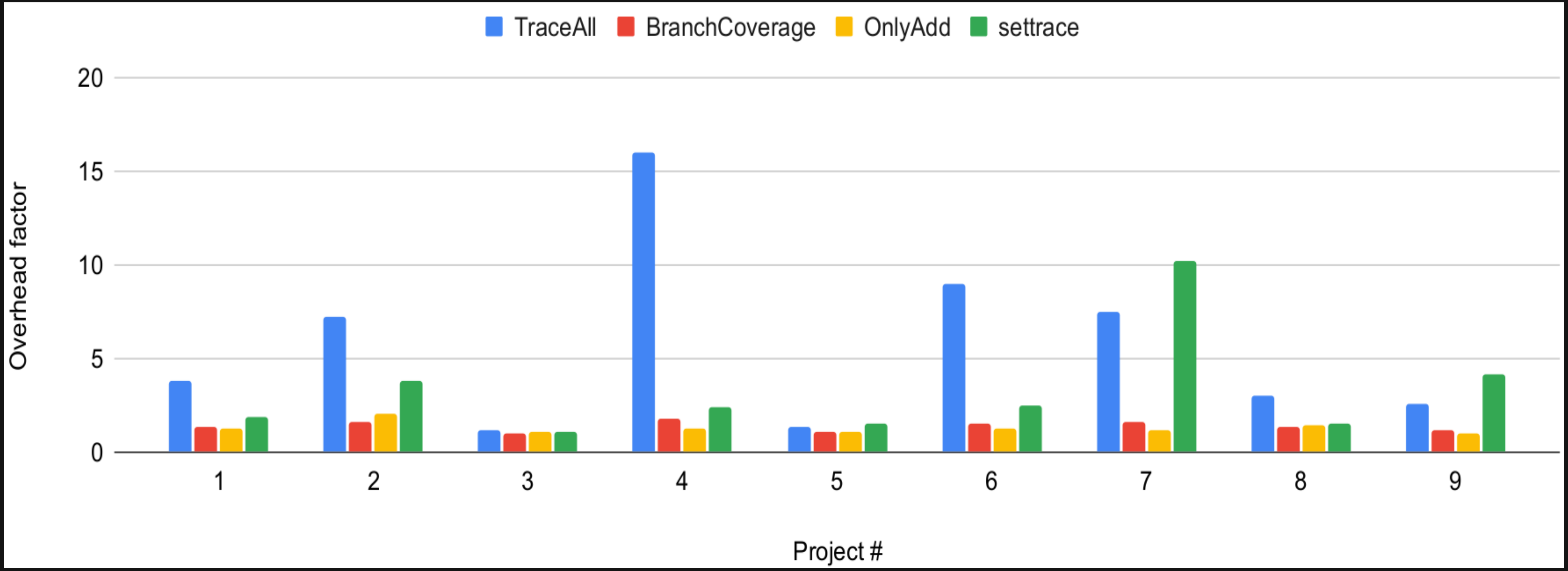


Runtime Overhead



Runtime Overhead

Built-in Python API



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

Talk to me about analysis ideas!