

Towards Effective Static Type-Error Detection for Python

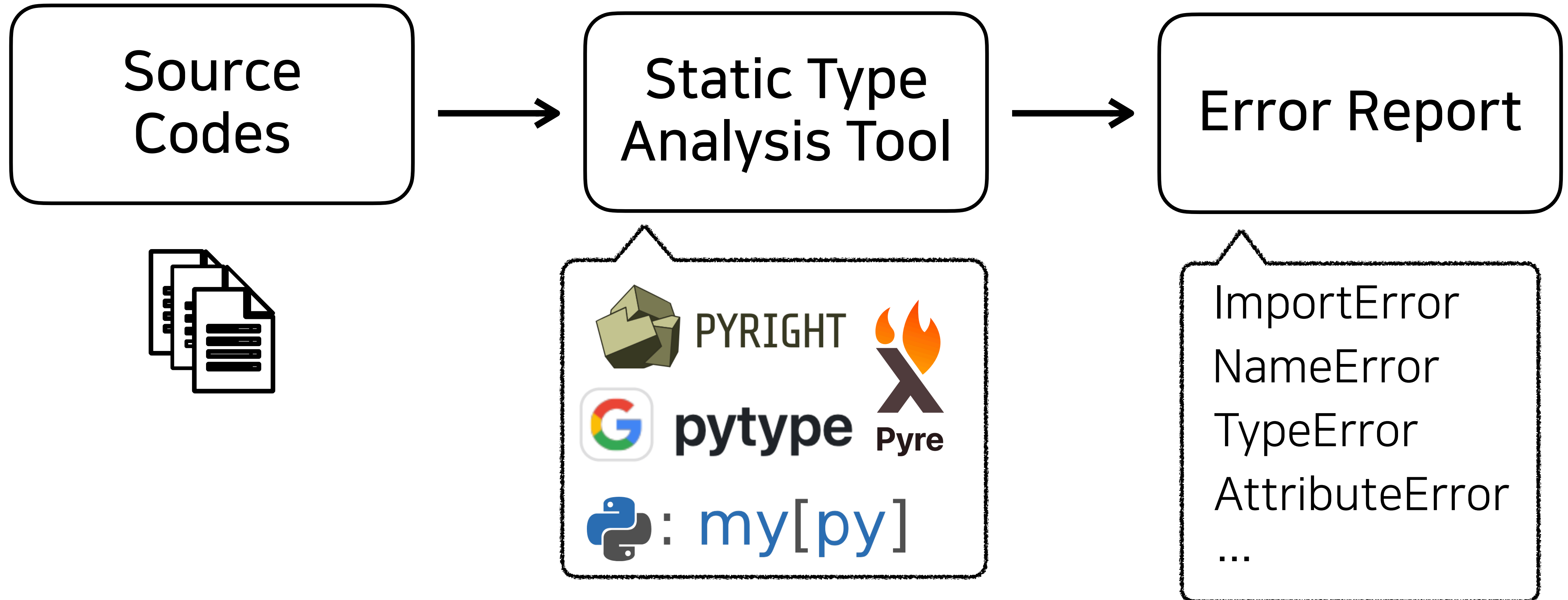
[Wonseok Oh](#)

Hakjoo Oh*

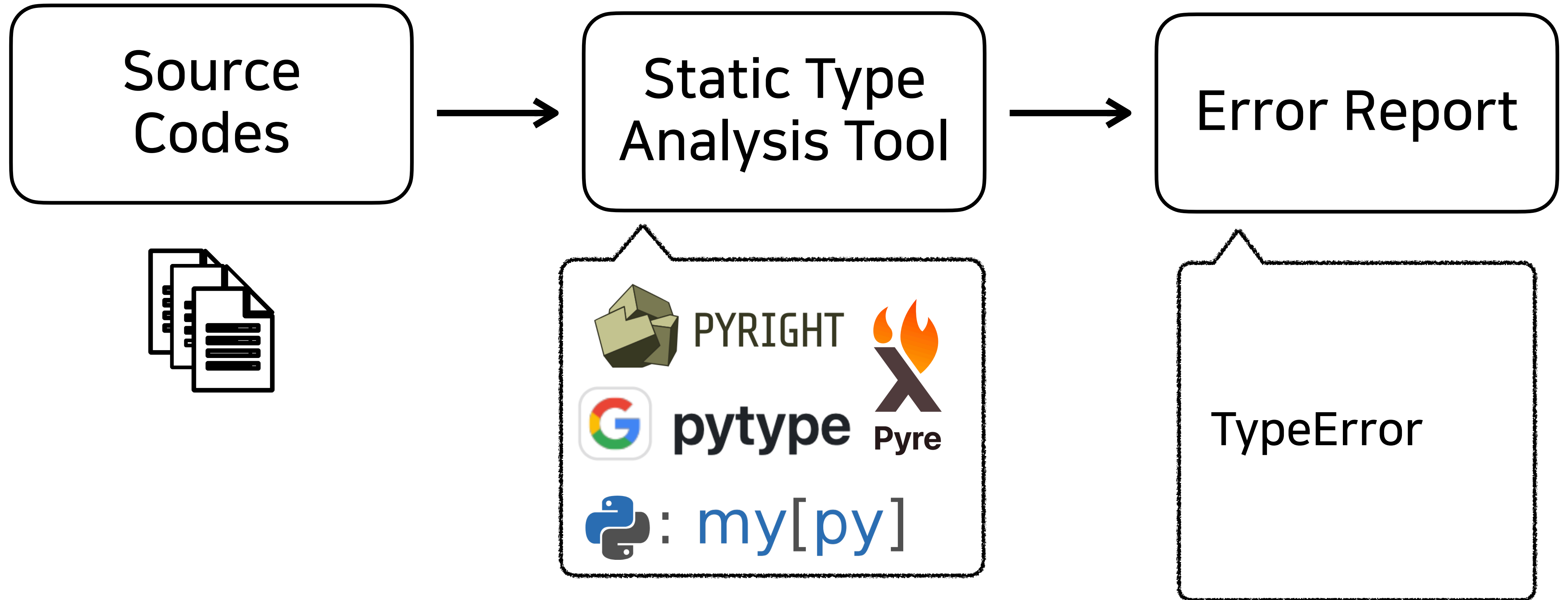


ASE 2024 @ Sacramento, California, US

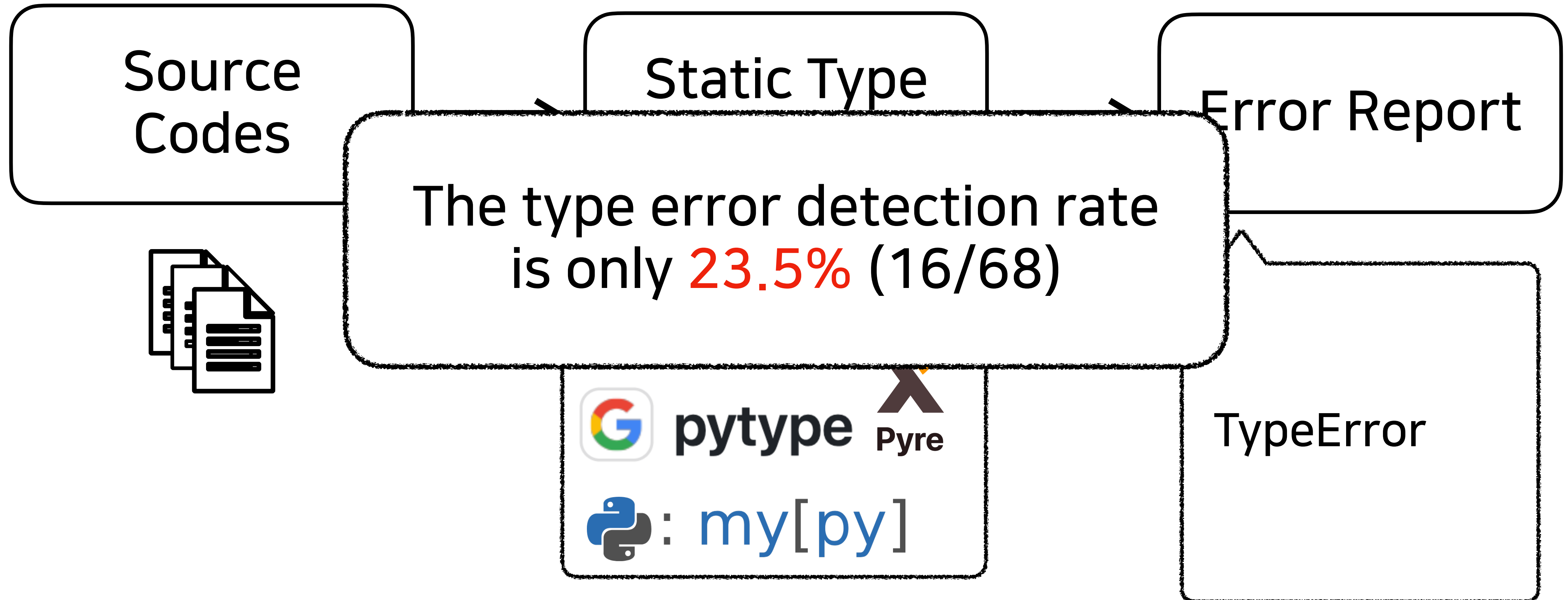
Introduction



Introduction



Introduction



Type Error

- “**type error**” denotes either ...
 - a runtime `TypeError` exception or
 - a `type mismatch` between type annotations and actual values

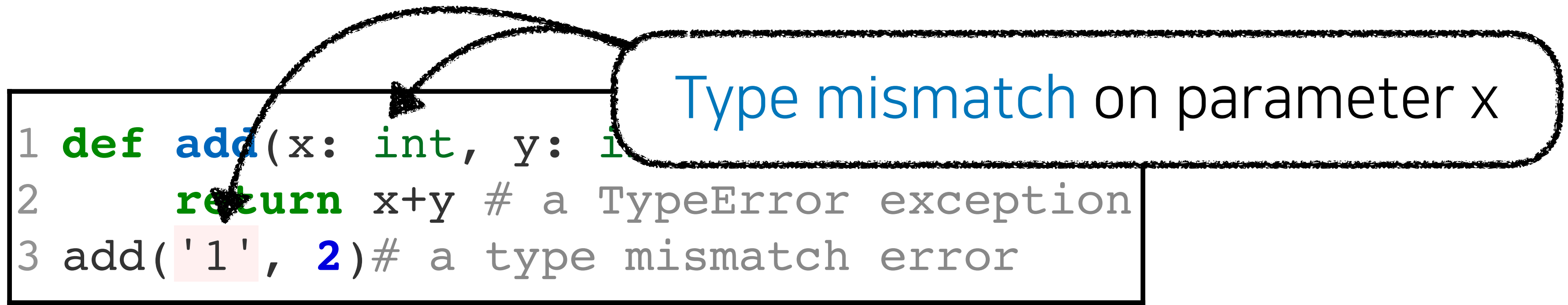
```
1 def add(x: int, y: int):  
2     return x+y # a TypeError exception  
3 add('1', 2) # a type mismatch error
```

Running this code causes `type error exception` at line 2

Type Error

- “type error” denotes either ...
 - a runtime `TypeError` exception or
 - a `type mismatch` between type annotations and actual values

```
1 def add(x: int, y: int) -> int:  
2     return x+y # a TypeError exception  
3 add('1', 2) # a type mismatch error
```

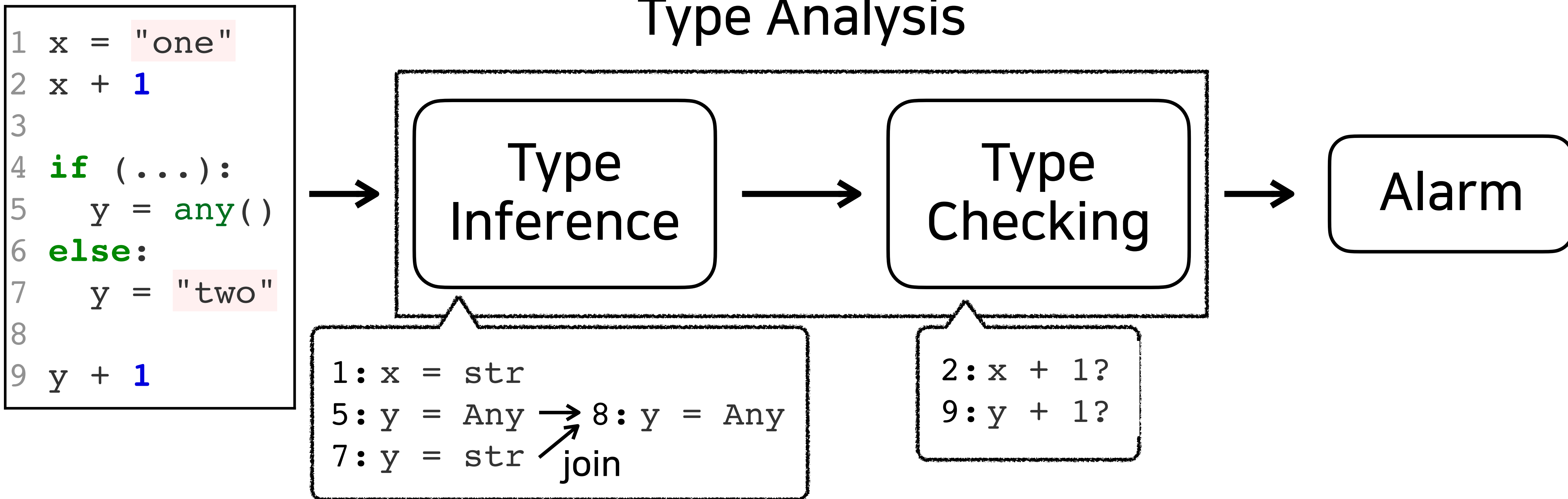


Type mismatch on parameter x

Running this code causes `type error exception` at line 2

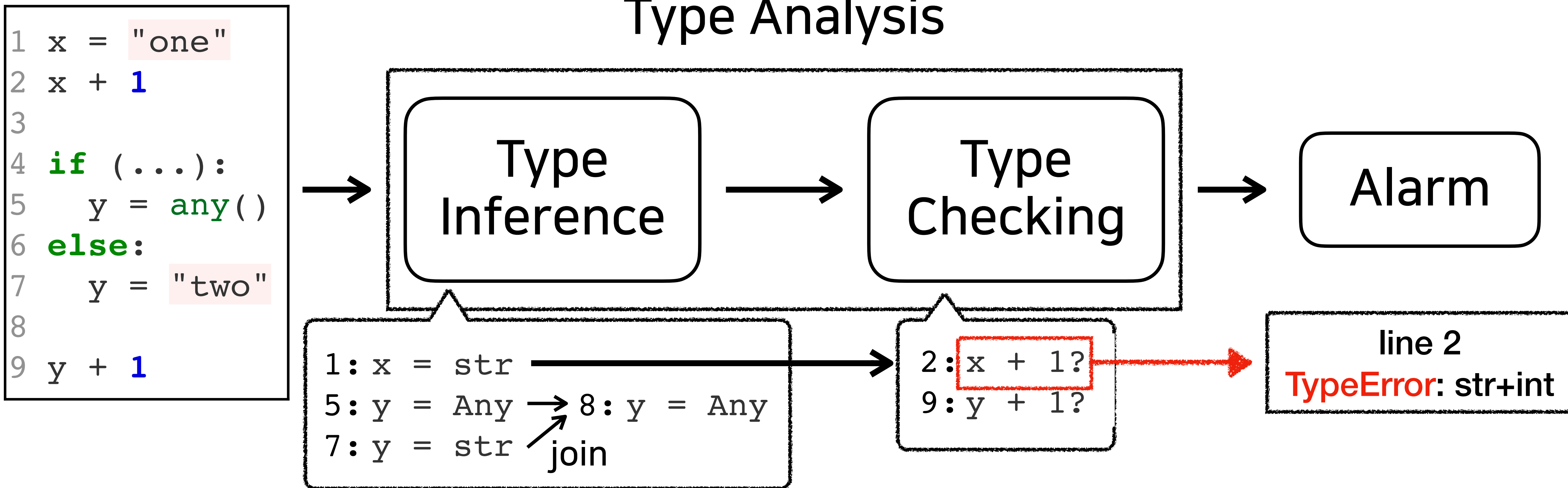
Overview

Type Analysis



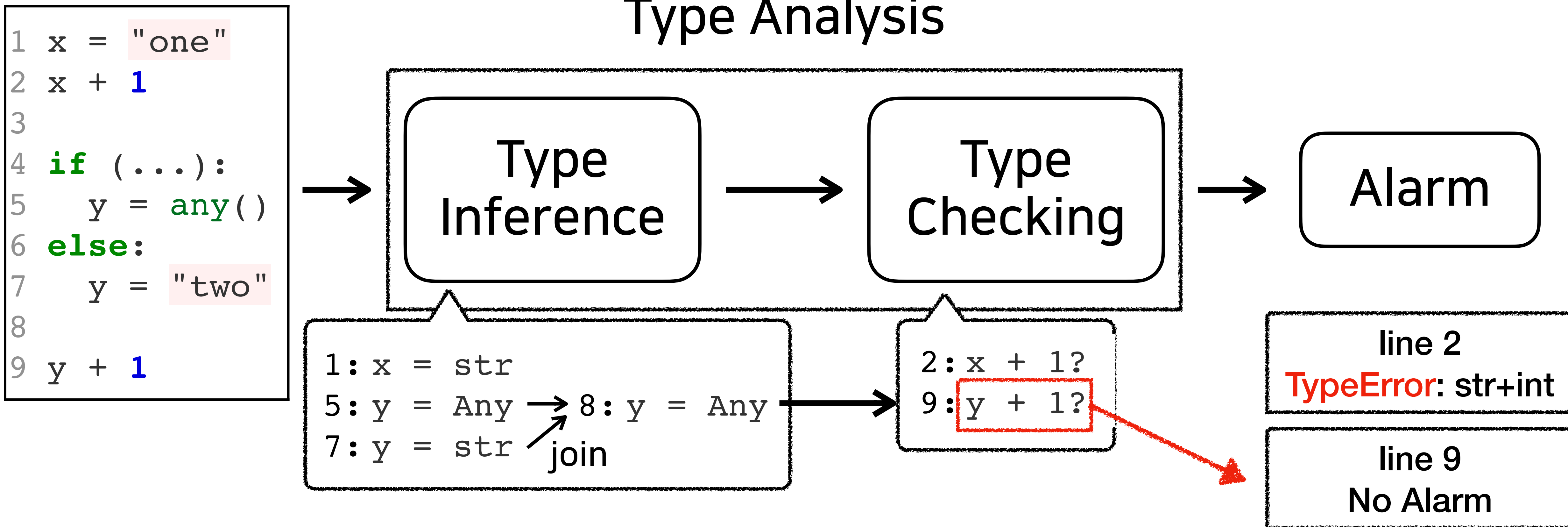
Overview

Type Analysis



Overview

Type Analysis



Problem

if *Any* type is included in alarm reports...

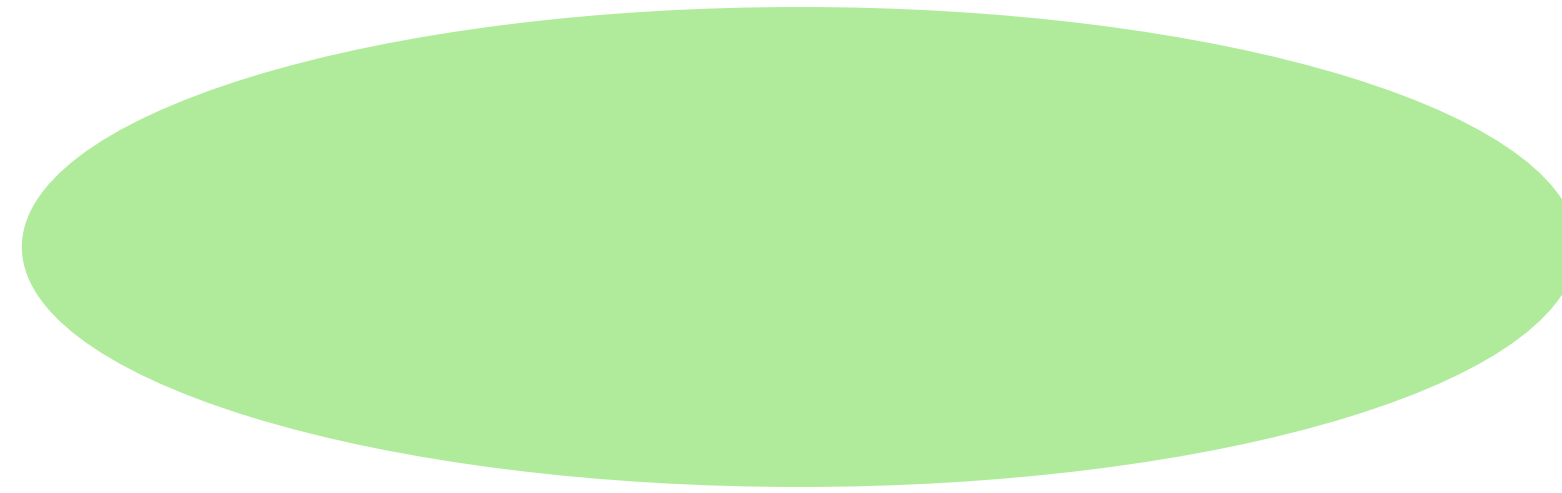
Problem

if *Any* type is included in alarm reports...

sound but too *imprecise*

Problem

Current Status of Existing Tools



fewer false alarms,
but **many type errors** are being **missed**

Problem

Current Status of Existing Tools

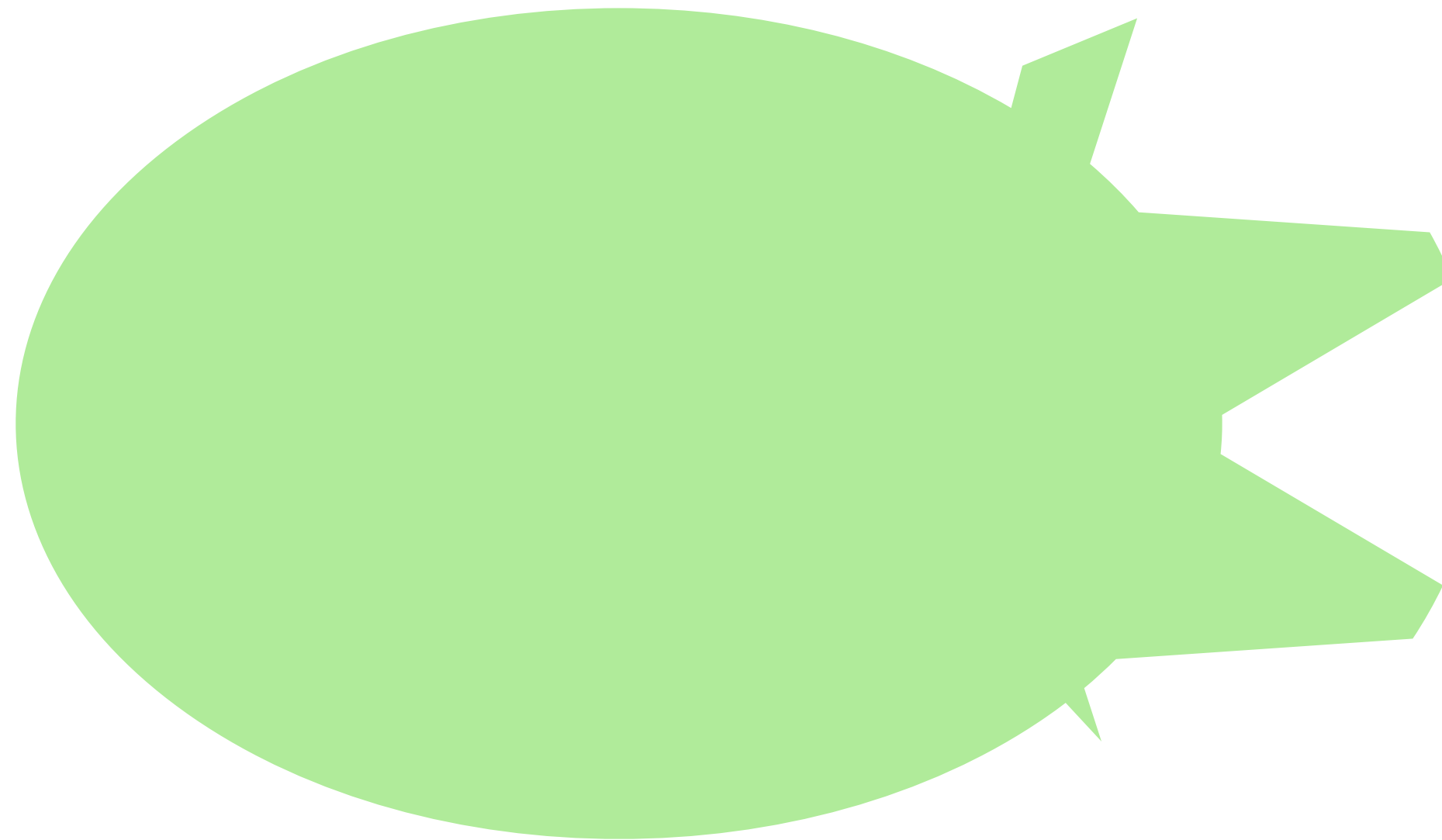


The type error detection rate
is only **23.5%** (16/68)

fewer false alarms,
but **many type errors** are being **missed**

Problem

Our Goal



How can we detect **more** type errors **effectively**?

Problem

Type Analysis

```
1 x = "one"
2 x + 1
3
4 if (...):
5     y = any()
6 else:
7     y = "two"
8
9 y + 1
```

Type
Inference

How can we improve
type inference performance
to better detect type errors?

```
1: x = str
5: y = Any → 8: y = Any
7: y = str   join
```

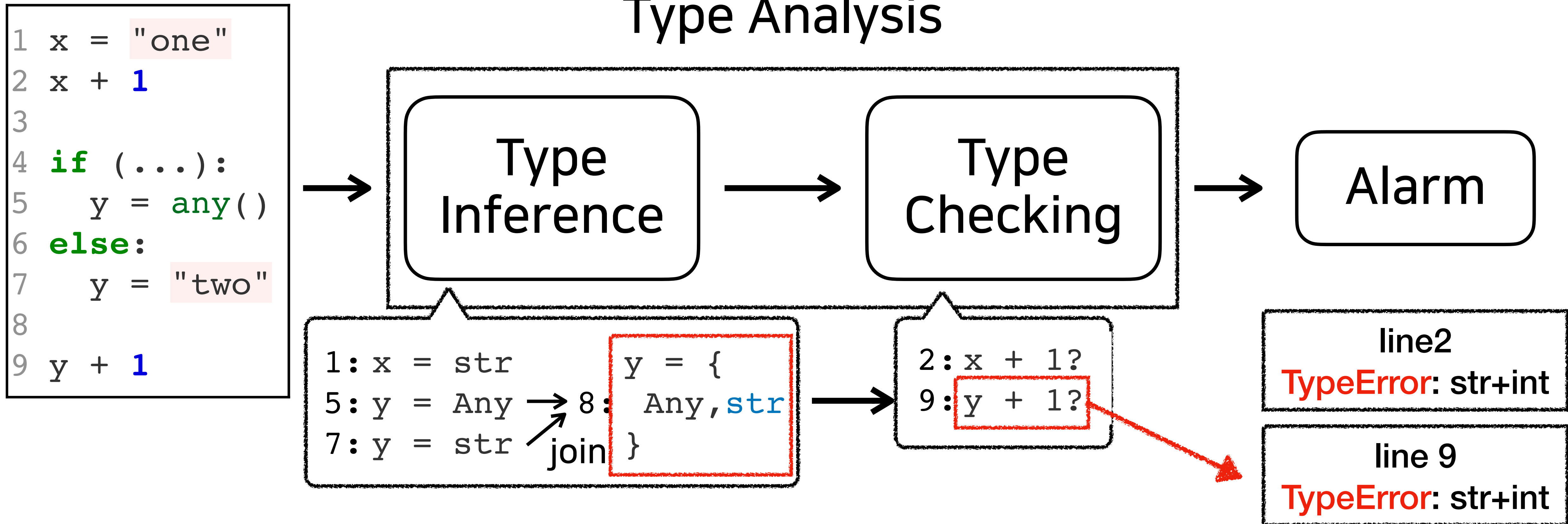
9: y + 1?

~~TypeError~~. str+int

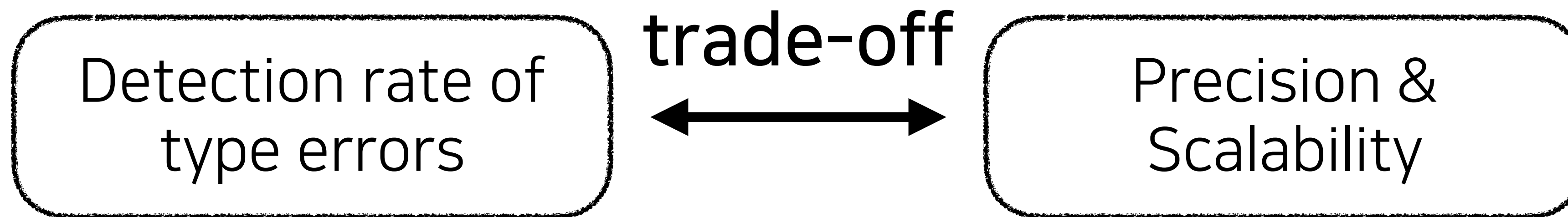
line 9
No Alarm

Problem

Type Analysis



Challenge



- We investigated real-world type errors to find features that ...
 - maximize the detection rate
 - minimize the negative effects on precision & scalability

Key Features

- We found **four main features** that are essential for the effective type error detection
 - Preserving Types when Merging
 - Cost-Effective Interprocedural Analysis
 - Usage-based Inference of Likely Types
 - Inference of Intended Member Types

Preserving Types when Merging

To maximize detection rate...

- **Preserve types** instead of merging types into Any

```
1  if (...):  
2      y = any()  
3  else:  
4      y = "two"  
5  # TypeError: int+str  
6  y + 1
```

Control Flow
Merge points

y = Any y = str
 ↓ ↓
 ~~y = Any~~
y = { Any, str }

Preserving Types when Merging

To maximize detection rate...

- **Preserve types** instead of merging types into Any

```
1  if (...):  
2      y = any()  
3  else:  
4      y = "two"  
5  # TypeError: int+str  
6  y + 1
```

Control Flow
Merge points

y = Any y = str
↓ ↓
~~y = Any~~
y = { Any, str }

```
1  x = [1, '2']  
2  # TypeError: int+str  
3  x[0] + "two"
```

Collections

~~x = List[Any]~~
x = List[int | str]

Preserving Types when Merging

To maximize detection rate...

- **Preserve types** instead of merging types into Any

```
1  if (...):  
2      y = any()  
3  else:  
4      y = "two"  
5  # TypeError: int+str  
6  y + 1
```

Control Flow
Merge points

43% of type errors require

y = Any y = str
↓ ↓
~~y = Any~~
y = { Any, str }

```
1  x = [1, '2']  
2  # TypeError: int+str  
3  x[0] + "two"
```

Collections

30% of type errors require

~~x = List[Any]~~
x = List[int | str]

Preserving Types when Merging

To maximize detection rate...

- **Preserve types** instead of merging types into Any

Negative Effects

- Increase **false alarms** by **dictionary**

```
1 x = {1: 1, "2": "2"}  
2 # TypeError: int+str  
3 x[1] + "1"  
4 x[1] + 1
```

`x = Dict[ int|str → int|str ]`

Preserving Types when Merging

To maximize detection rate...

- **Preserve types** instead of merging types into Any

Negative Effects

- Increase **false alarms** by **dictionary**

```
1 x = {1: 1, "2": "2"}
2 # TypeError: int+str
3 x[1] + "1"
4 x[1] + 1
```

x = Dict[int|str → int|str]

False Alarm
TypeError: str+int

Preserving Types when Merging

To maximize detection rate...

- **Preserve types** instead of merging types into Any

Negative Effects

- Increase **false alarms** by **dictionary**

To minimize negative effects...

- Consider **key-type sensitive dictionary**

```
1 x = {1: 1, "2": "2"}
2 # TypeError: int+str
3 x[1] + "1"
4 x[1] + 1
```



x = Dict[int|str → int|str]



False Alarm
TypeError: str+int

Preserving Types when Merging

To maximize detection rate...

- Preserve types instead of merging types into Any

Negative Effects

- Increase false alarms by dictionary

To minimize negative effects...

- Consider key-type sensitive dictionary

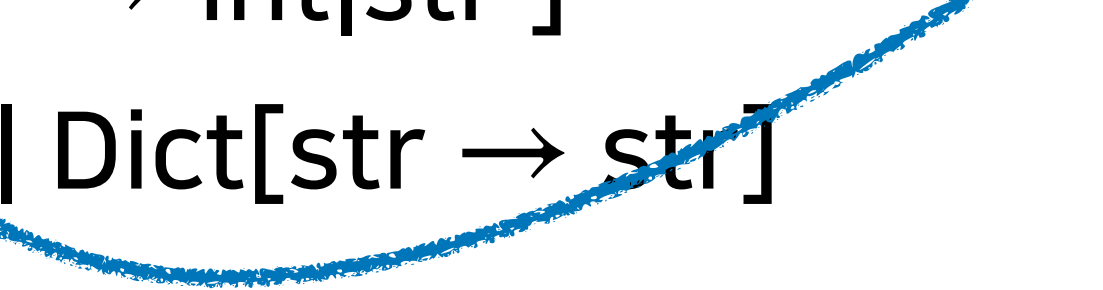
```
1 x = {1: 1, "2": "2"}
2 # TypeError: int+str
3 x[1] + "1"
4 x[1] + 1
```



No Alarm!

~~x = Dict[int|str → int|str]~~

x = Dict[int → int] | Dict[str → str]



Preserving Types when Merging

To maximize detection rate...

- Preserve types instead of merging types into Any

A quarter of the alarms are eliminated,
all of which were false alarms

```
1 x = {1: 1, "2": "2"}  
2 # TypeError: int+str  
3 x[1] + "1"  
4 x[1] + 1
```

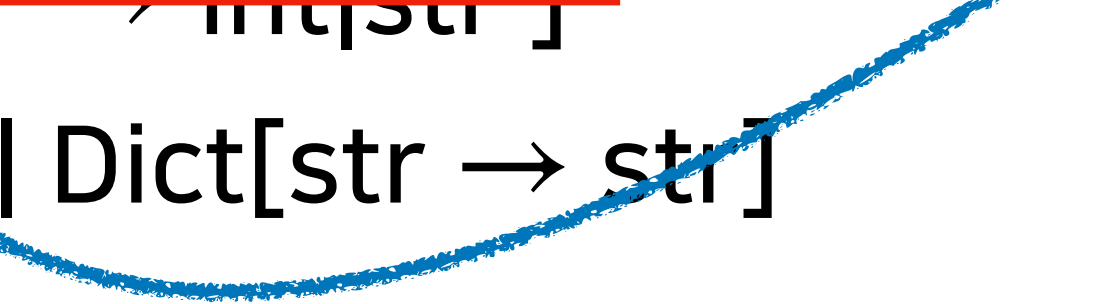


No Alarm!

To minimize negative effects...

~~x = Dict[int|str → int|str]~~

x = Dict[int → int] | Dict[str → str]



- Consider key-type sensitive dictionary

Cost-Effective Interprocedural Analysis

To maximize detection rate...

- About 54% type errors require interprocedural analysis

Negative Effects

- Blow up the analysis cost...

Cost-Effective Interprocedural Analysis

To maximize detection rate...

- About 54% type errors require interprocedural analysis

Negative Effects

- Blow up the analysis cost...

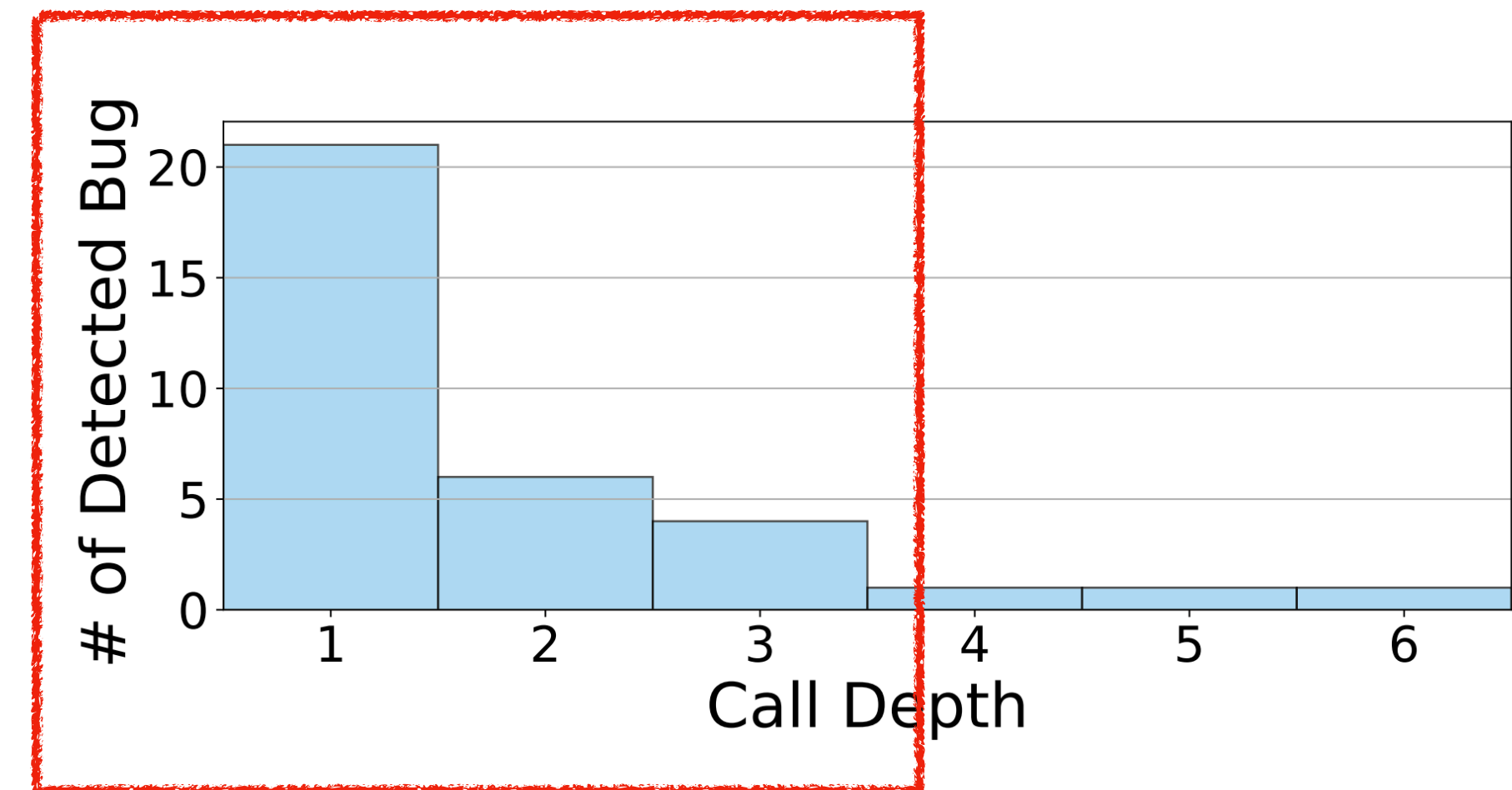


Figure 2: Call depth required to detect type errors.

Approximately 84% can be detected with a shallow call depth (≤ 3)

Cost-Effective Interprocedural Analysis

To maximize detection rate...

- About 54% type errors require interprocedural analysis

Negative Effects

- Blow up the analysis cost...

To minimize negative effects...

- Apply shallow call depth

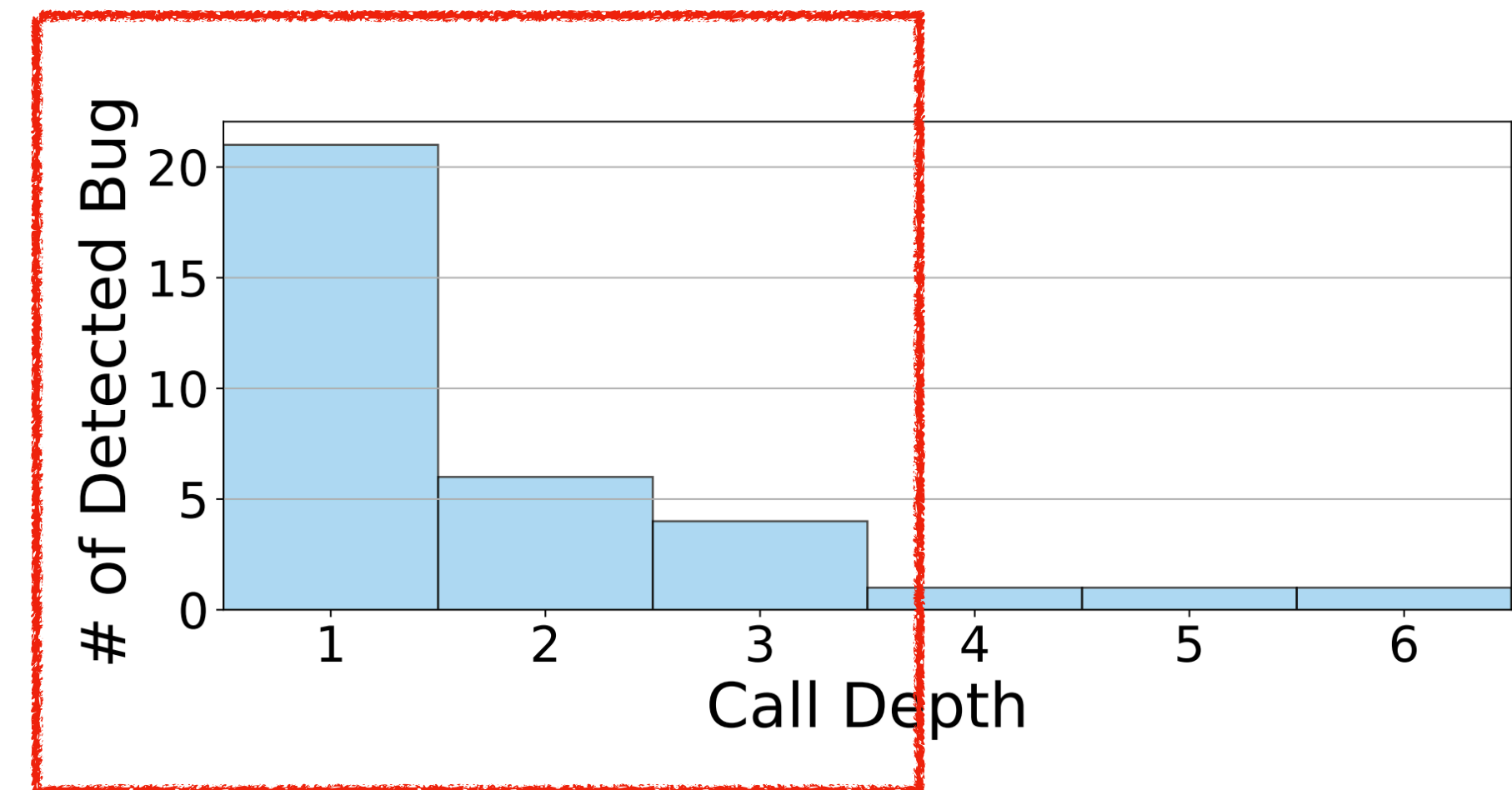


Figure 2: Call depth required to detect type errors.

Approximately 84% can be detected with a shallow call depth (≤ 3)

Cost-Effective Interprocedural Analysis

To maximize detection rate...

- About 54% type errors require interprocedural analysis

Reduce the cost by 78%

Blow up the analysis cost

To minimize negative effects...

- Apply shallow call depth

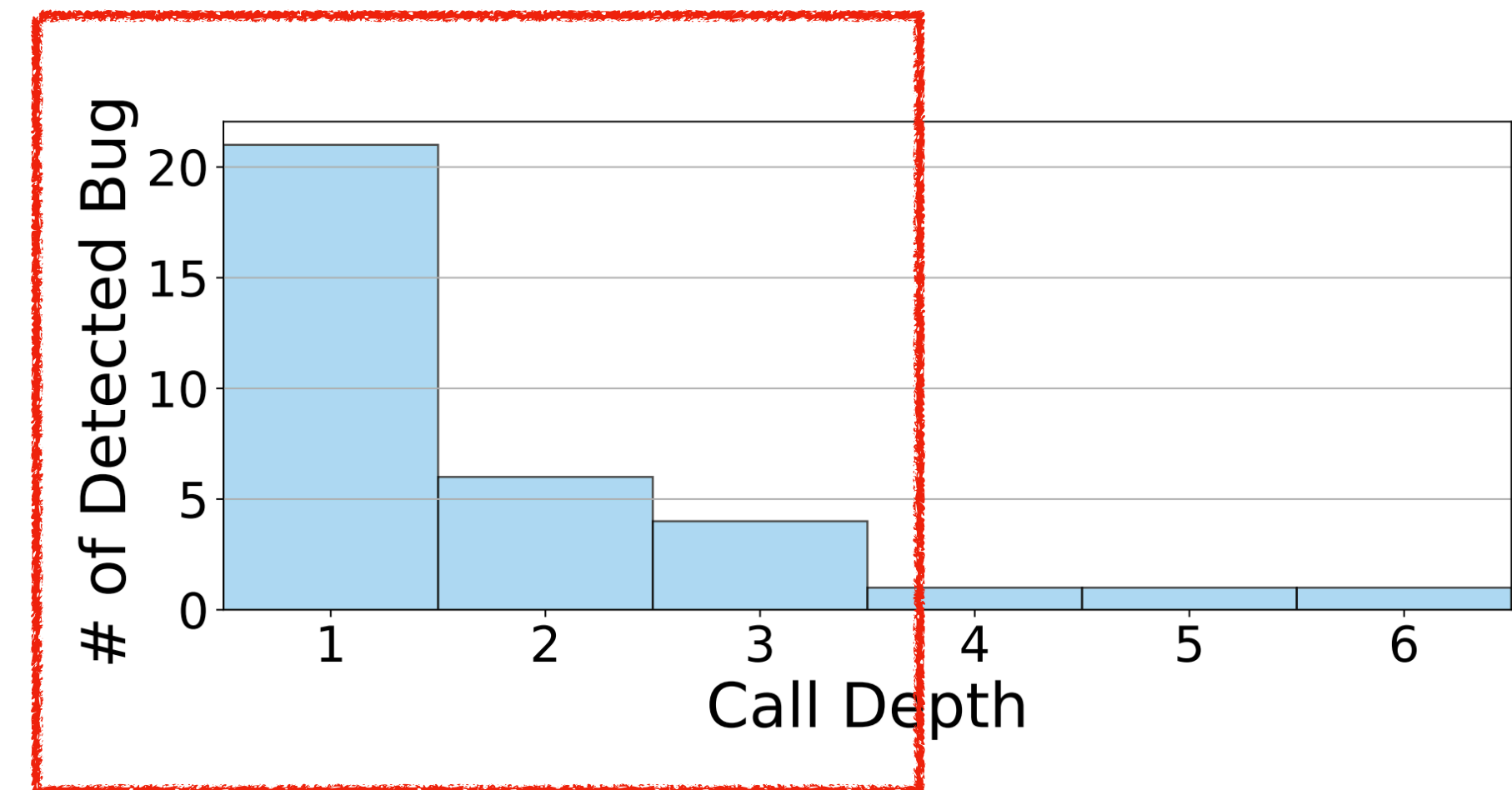


Figure 2: Call depth required to detect type errors.

Approximately 84% can be detected with a shallow call depth (≤ 3)

Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

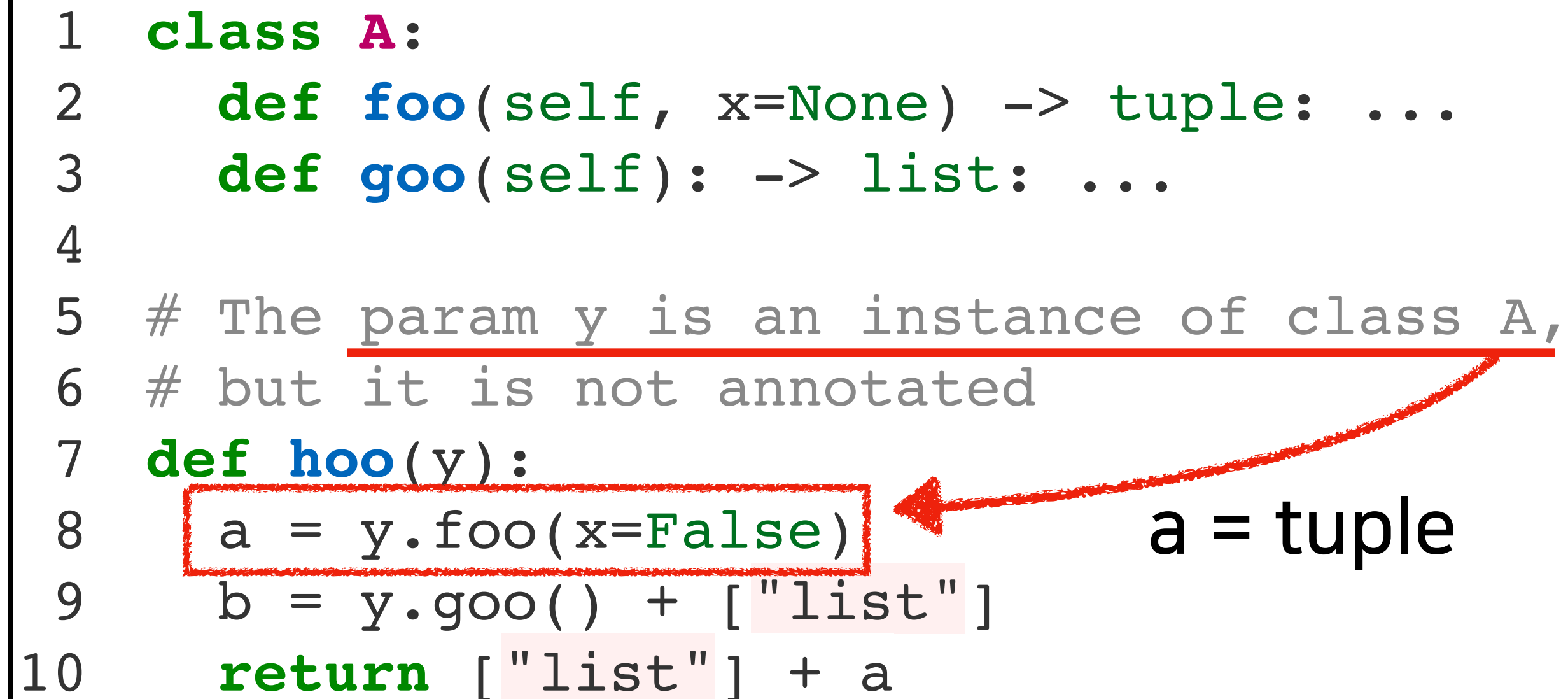
```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ["list"]
10     return ["list"] + a
```

Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ["list"]
10     return ["list"] + a
```



Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ["list"]
10     return ["list"] + a
```

TypeError: list+tuple

Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8
9
10     return ["list"] + a
```

How can we infer "y = A"?

Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ["list"]
10     return ["list"] + a
```

y is intended to be used
as an instance of class A

call these methods
without errors

y has two methods:
foo, goo

Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

Negative Effects

- Ineffective in practice because it often infers infeasible types

```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ["list"]
10     return ["list"] + a
11
12 class B:
13     def foo(self, **kwargs) -> list: ...
14     def goo(self) -> tuple: ...
```

Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

Negative Effects

- Ineffective in practice because it often infers infeasible types

```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ['list']
10     return ['list'] + ...
11
12 class B:
13     def foo(self, **kwargs) -> list: ...
14     def goo(self) -> tuple: ...
```

Annotations and Callouts:

- Line 8: `y.foo(x=False)` is boxed in blue.
- Line 9: `y.goo()` is boxed in blue.
- Line 10: `['list']` is highlighted in pink.
- Line 10: `return` is highlighted in green.
- Line 10: `['list']` is highlighted in pink.
- Line 10: `+` is highlighted in pink.
- Line 10: `...` is highlighted in pink.
- Line 10: `call these methods without errors` is written in blue.
- Line 10: `y.foo(...) = list` is written in blue.
- Line 10: `y.goo() = tuple` is written in blue.

Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

Negative Effects

- Ineffective in practice because it often infers infeasible types

```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ["list"]
10     return ["list"] + a

```

y.foo(...) = list
y.goo() = tuple

Missing type error

```
11 class B:
12     def foo(self, **kwargs) -> list: ...
13     def goo(self) -> tuple: ...

```


Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

Negative Effects

- Ineffective in practice because it often infers infeasible types

```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ["list"]
10     return ["list"] + a

```

y.foo(...) = list
y.goo() = tuple

```
11 class B:
12     def foo(self, **kwargs) -> list: ...
13     def goo(self) -> tuple: ...

```

Generate False Alarm

Usage-based Inference of Likely Types

To maximize detection rate...

- Infer **types** of function parameters **by observing how they are used**

Negative Effects

- **Ineffective in practice** because it often infers **infeasible types**

To minimize negative effects...

- Extract **likely type** instead of using all types

```
1  class A:
2      def foo(self, x=None) -> tuple: ...
3      def goo(self): -> list: ...
4
5  # The param y is an instance of class A,
6  # but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ["list"]
10     return ["list"] + a
11
12 class B:
13     def foo(self, **kwargs) -> list: ...
14     def goo(self) -> tuple: ...
```

Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

Negative Effects

- Ineffective in practice
it often infers infeasible types

To minimize negative effects...

- Extract likely type instead of using all types

`foo` is called in a way
that is more similar to class A

an instance of class A,
annotated

```
1 class A:
2     def foo(self, x=None) -> tuple: ...
3     def goo(self): -> list: ...
```

```
7 def foo(y):
8     a = y.foo(x=False)
9     b = y.goo() + ["list"]
10    return ["list"] + a
```

```
11 class B:
12     def foo(self, **kwargs) -> list: ...
13     def goo(self) -> tuple: ...
```


Usage-based Inference of Likely Types

To maximize detection rate...

- Infer types of function parameters by observing how they are used

This way is 2.7 times more efficient
and produces 10% fewer alarms

it often infers infeasible types

To minimize negative effects...

- Extract likely type instead of using all types

```
1  class A:
    def foo(self, x=None) -> tuple: ...
    def goo(self): -> list: ...

# The param y is an instance of class A,
# but it is not annotated
7  def hoo(y):
8      a = y.foo(x=False)
9      b = y.goo() + ["list"]
10     return ["list"] + a

11 class B:
12     def foo(self, **kwargs) -> list: ...
13     def goo(self) -> tuple: ...
```


Inference of Intended Member Types

To maximize detection rate...

- Consider **all possible method call scenarios** to detect bugs

```
1  class A():
2      def __init__(self, x: Any): self.x = x
3      def set_x(self, x=None):    self.x = x
4      def foo(self, v: int):
5          if v < self.x: # TypeError: int < None
6              ...
```

TypeError

$a = A(\dots) \rightarrow a.set_x() \rightarrow a.foo(1)$

Inference of Intended Member Types

To maximize detection rate...

- Consider **all possible method call scenarios** to detect bugs

```
1  class A():
2      def __init__(self, x: Any): self.x = x
3      def set_x(self, x=None):    self.x = x
4      def foo(self, v: int):
5          if v < self.x: # TypeError: int < None
6              ...
```

TypeError

$a = A(\dots) \rightarrow a.set_x() \rightarrow a.foo(1)$

We have to compute **class invariants** in terms of **member types**
such as "The type of self.x can be None in class A"

Inference of Intended Member Types

To maximize detection rate...

- Consider **all possible method call scenarios** to detect bugs

Negative Effects

- Generate **false alarms** by **infeasible method call scenarios**

```
1  class B():
2      def __init__(self):      self.x = None
3      def setup(self, x: int): self.x = x
4      def foo(self, y: int):
5          return x+y
6
7  def use_B():
8      b = B()
9      b.setup(1)
10     b.foo(2) # No TypeError
```

Inference of Intended Member Types

To maximize detection rate...

- Consider all possible method call scenarios

$b = B() \rightarrow b.foo(2)$
 $\Rightarrow$ Generate **TypeError** alarm

Negative Effects

- Generate **false alarms** by **infeasible method call scenarios**

```
1  class B():
2      def __init__(self):          self.x = None
3      def setup(self, x: int):      self.x = x
4      def foo(self, y: int):
5          return x+y
6
7  def use_B():
8      b = B()
9      b.setup(1)
10     b.foo(2) # No TypeError
```

Inference of Intended Member Types

To maximize detection rate...

- Consider **all possible method call scenarios**

$b = B() \rightarrow b.foo(2)$
 $\Rightarrow$ Generate **TypeError** alarm

Negative Effects

- Generate **false alarms** by **infeasible method call scenarios**

Class B is **intended** to use **foo** **after calling setup**

```
1  class B():
2      def __init__(self):          self.x = None
3      def setup(self, x: int):     self.x = x
4      def foo(self, y: int):
5          return x+y
6
7  def use_B():
8      b = B()
9      b.setup(1)
10     b.foo(2) # No TypeError
```


Inference of Intended Member Types

To maximize detection rate...

- Consider **all possible method call scenarios**

b = B().foo(2)
=> Generate **TypeError** alarm

Negative Effects

- Generate **false alarms**
infeasible method call scenarios

This scenario **was not considered**
by developers

Class B is **intended** to use
foo **after calling** **setup**

```
1 class B():
2     def setup(self, x):
3         self.x = None
4         self.x = x
5         return x+y
6
7 def use_B():
8     b = B()
9     b.setup(1)
10    b.foo(2) # No TypeError
```


Inference of Intended Member Types

To maximize detection rate...

- Consider **all possible method call scenarios** to detect bugs

Negative Effects

- Generate **false alarms** by **infeasible method call scenarios**

```
1  class B():
2      def __init__(self):          self.x = None
3      def setup(self, x: int):    self.x = x
4      def foo(self, y: int):
5          return x+y # type: ignore
6
7  def us
8      b = B()
9      b.setup(1)
10     b.foo(2) # No TypeError
```

Use alarm ignore comments...

Inference of Intended Member Types

To maximize detection rate...

- Consider **all possible method call scenarios** to detect bugs

Negative Effects

- Generate **false alarms** by **infeasible method call scenarios**

To minimize negative effects...

- Distinguish** infeasible scenarios

```
1  class B():
2      def __init__(self):          self.x = None
3      def setup(self, x: int):    self.x = x
4      def foo(self, y: int):
5          return x+y # type: ignore
6
7  def use_B():
8      b = B()
9      b.setup(1)
10     b.foo(2) # No TypeError
```


Inference of Intended Member Types

To maximize detection rate...

- Consider **all possible method call scenarios** to detect bugs

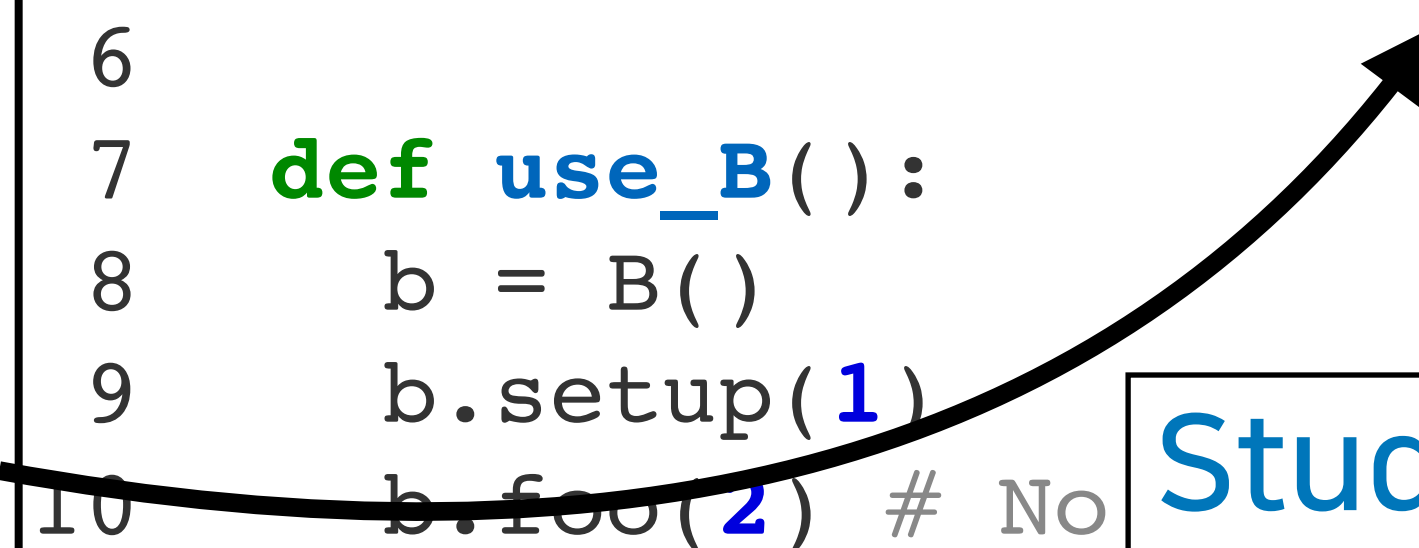
Negative Effects

- Generate **false alarms** by **infeasible method call scenarios**

To minimize negative effects...

- Distinguish** infeasible scenarios

```
1  class B():
2      def __init__(self):      self.x = None
3      def setup(self, x: int): self.x = x
4      def foo(self, y: int):
5          return x+y # type: ignore
6
7  def use_B():
8      b = B()
9      b.setup(1)
10     b.foo(2) # No
```



Study these cases and
use **learning algorithm**

Key Features

- We found that **four main features** are essential for the type error detection
 - Preserving Types when Merging
 - Cost-Effective Interprocedural Analysis
 - Usage-based Inference of Likely Types
 - Inference of Intended Member Types

Key Features: Summary

Detailed points for supporting the feature		Mypy	Pyre	Pytype	Pyright	Pyinder
P1	Preserving types at control flow merge points	×	×	×	○	○
P2	Preserving element types in collections	×	×	○	○	○
P3	Considering dictionaries with a key-type-sensitive manner	×	×	×	×	○
P4	Shallow call depths (up to 3)	×	×	×	×	○
P5	Lightweight type-based context sensitivity	×	×	○	×	○
P6	Usage-based type inference	×	×	×	×	○
P7	Selecting likely types among usage-based types	×	×	×	×	○
P8	Considering class invariants in terms of member types	×	×	×	○	○
P9	Filtering out unintended member types	×	×	×	×	○

- Preserving Types when Merging (**P1, P2, P3**)
- Cost-Effective Interprocedural Analysis (**P4, P5**)
- Usage-based Inference of Likely Types (**P6, P7**)
- Inference of Intended Member Types (**P8, P9**)

Evaluation

- **Effectiveness**

How effectively does Pyinder detect type errors in the benchmarks?

- **Ablation study**

How does each feature of Pyinder contribute to the final performance?

- **Generality**

Can Pyinder detect critical type errors in unseen, recently developed open-source Python projects?

Benchmark

- We collected 145 type errors from the three benchmarks
 - TypeBugs (FSE'22), BugsInPy (FSE'20), and ExcePy (SANER'22)
- We categorized the type error into 4 types...

Benchmark

- We collected 145 type errors from the three benchmarks
 - TypeBugs (FSE'22), BugsInPy (FSE'20), and ExcePy (SANER'22)
- We categorized the type error into 4 types...
 - Internal Error

```
1 a = 1
2 b = '2'
3
4 # TypeError
5 a + b
```

Error location is within
the current Python project

Benchmark

- We collected 145 type errors from the three benchmarks
 - TypeBugs (FSE'22), BugsInPy (FSE'20), and ExcePy (SANER'22)
- We categorized the type error into 4 types...
 - Internal Error
 - External Error

```
1 import numpy as np
2
3 v = [1, None]
4 # TypeError
5 w = np.sort(v)
```

Error location is in
external libraries

Benchmark

- We collected 145 type errors from the three benchmarks
 - TypeBugs (FSE'22), BugsInPy (FSE'20), and ExcePy (SANER'22)
- We categorized the type error into 4 types...
 - Internal Error
 - External Error
 - User-defined errors

```
1 a = int_or_str()  
2  
3 if isinstance(a, int):  
4     # TypeError  
5     raise TypeError
```

Custom type errors explicitly
raised by developers

Benchmark

- We collected 145 type errors from the three benchmarks
 - TypeBugs (FSE'22), BugsInPy (FSE'20), and ExcePy (SANER'22)
- We categorized the type error into 4
 - Internal Error
 - External Error
 - User-defined Error
 - Cython Error

Cython

```
1 def add(x, y): # TypeError
2     cdef int ret = x + y
```

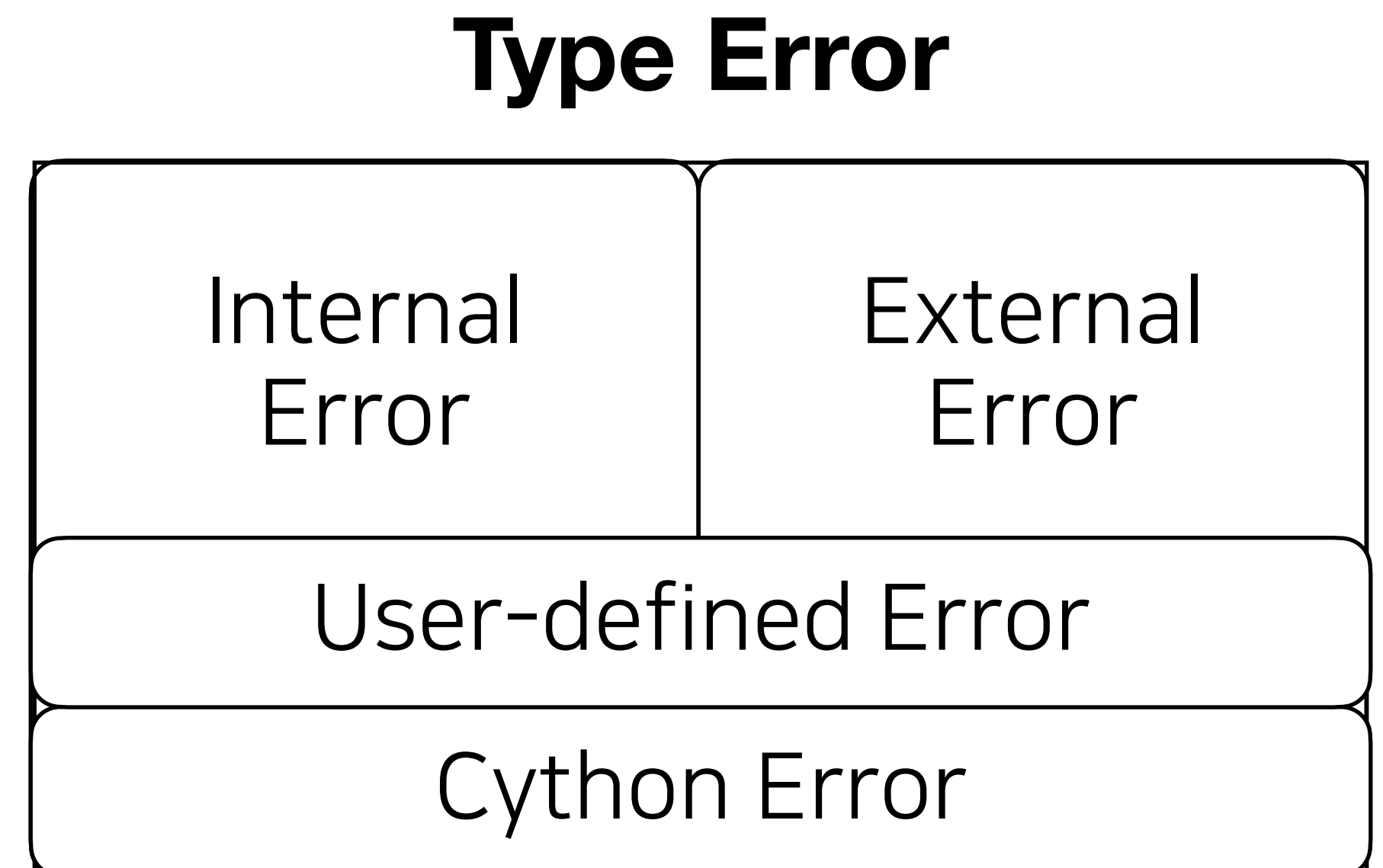
Python

```
1 a.add('1', '2')
```

Type errors raised in the
Cython language
(C extensions for Python)

Benchmark

- We collected 145 type errors from the three benchmarks
 - TypeBugs (FSE'22), BugsInPy (FSE'20), and ExcePy (SANER'22)
- We categorized the type error into 4 types...
 - Internal Error (42%)
 - External Error (23%)
 - User-defined Error (24%)
 - Cython Error (11%)



Benchmark

- We collected **145 type errors** from the three benchmarks
 - TypeBugs (FSE'22), BugsInPy (FSE'20), and ExcePy (SANER'22)
- We **categorized** the type error into **4 types**...

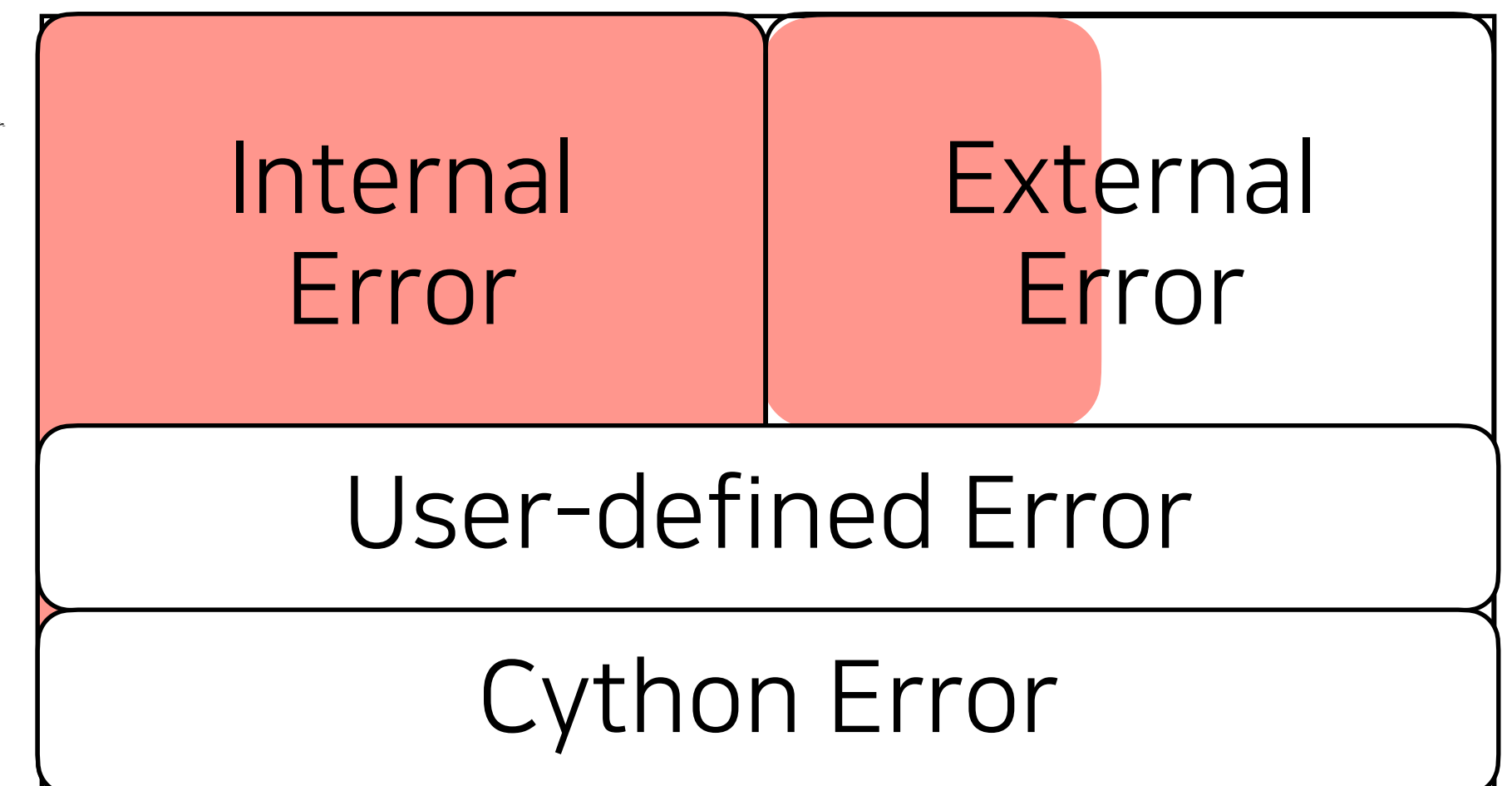
- Internal Error (42%)

- External Error (24%)
 - Chose **68 type errors** from 20 open-source projects (4k~417k lines) in this study

- User-defined Error (21%)

- Cython Error (11%)

Type Error



Benchmark

- We collected **145 type errors** from the three benchmarks
 - TypeBugs (FSE'22), BugsInPy (FSE'20), and ExcePy (SANER'22)
- We **categorized** the type error into **4 types**...

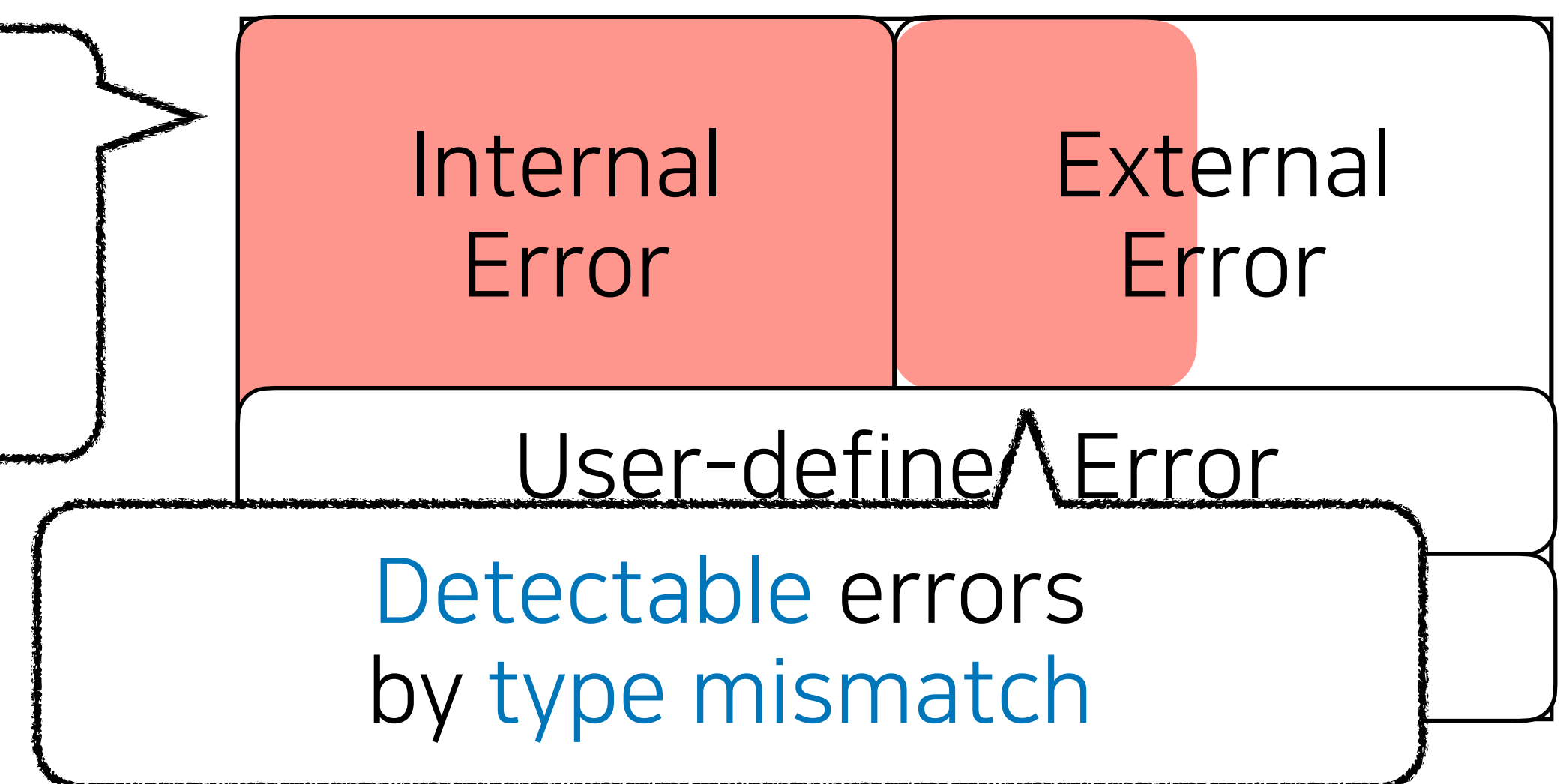
- Internal Error (42%)

- External Error (24%)
 - Chose **68 type errors** from 20 open-source projects (4k~417k lines) in this study

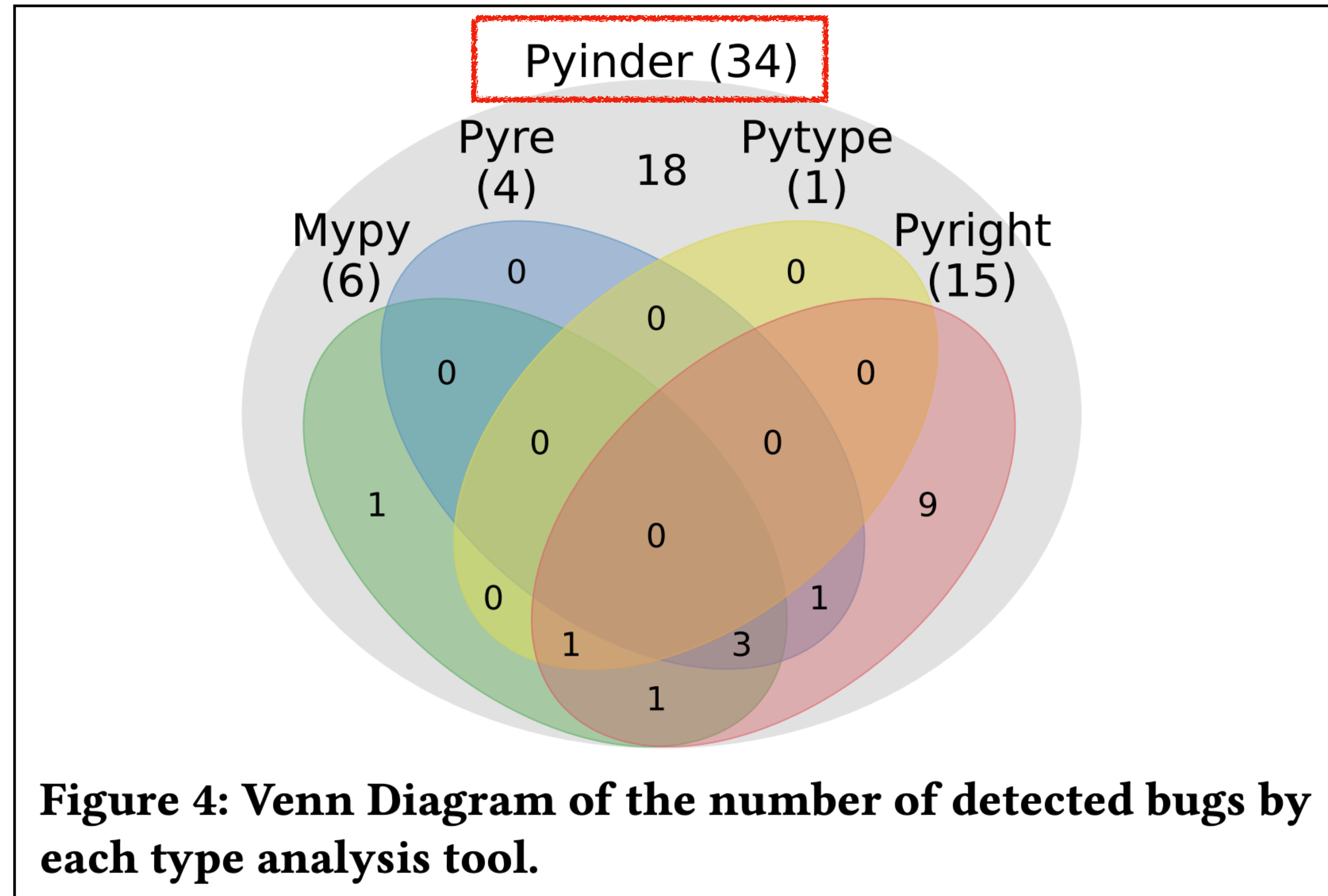
- User-defined Error (21%)

- Cython Error (11%)

Type Error



RQ1: Effectiveness



We totally uncovered **34 bugs (50%)** while other tools collectively detect **16 bugs**

RQ1: Effectiveness

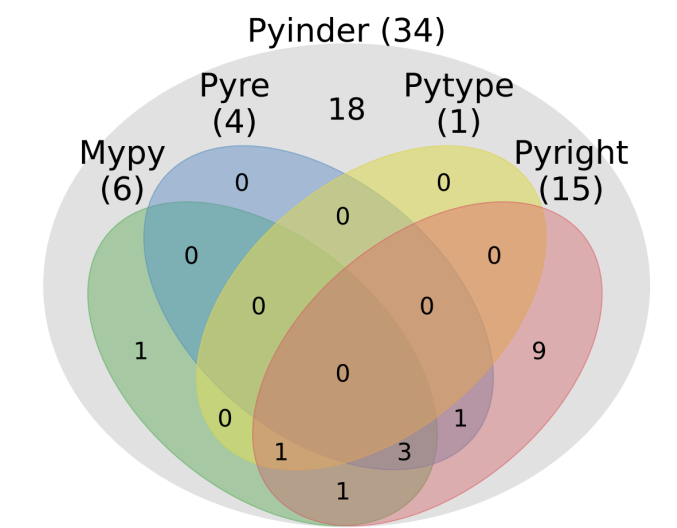


Figure 4: Venn Diagram of the number of detected bugs by each type analysis tool.

Table 2: Comparison with existing type analysis tools on the # of alarms and analysis time (both per 1KLoC).

	Mypy	Pyre	Pytype	Pyright	Pyinder
#alarms/KLoC	4.72	8.38	1.74	11.37	4.57
time(s)/KLoC	0.48	0.26	98.28	0.98	5.78

RQ1: Effectiveness

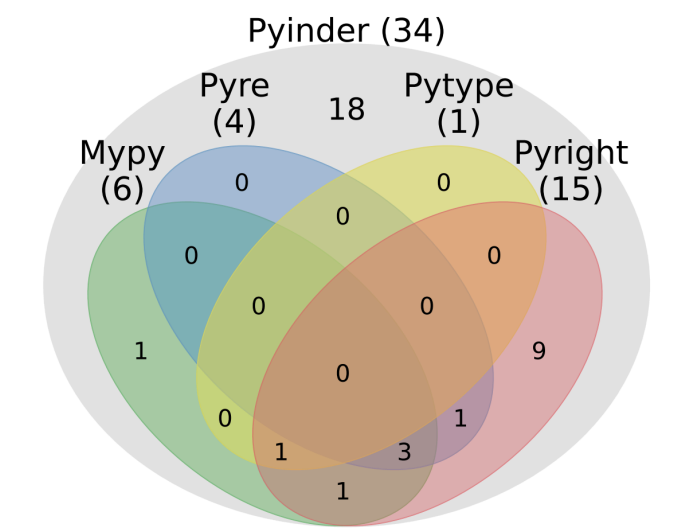


Figure 4: Venn Diagram of the number of detected bugs by each type analysis tool.

Table 2: Comparison with existing type analysis tools on the # of alarms and analysis time (both per 1KLoC).

	Mypy	Pyre	Pytype	Pyright	Pyinder
#alarms/KLoC	4.72	8.38	7.74	11.37	4.57
time(s)/KLoC	0.48	0.26	98.28	0.98	5.78

RQ1: Effectiveness

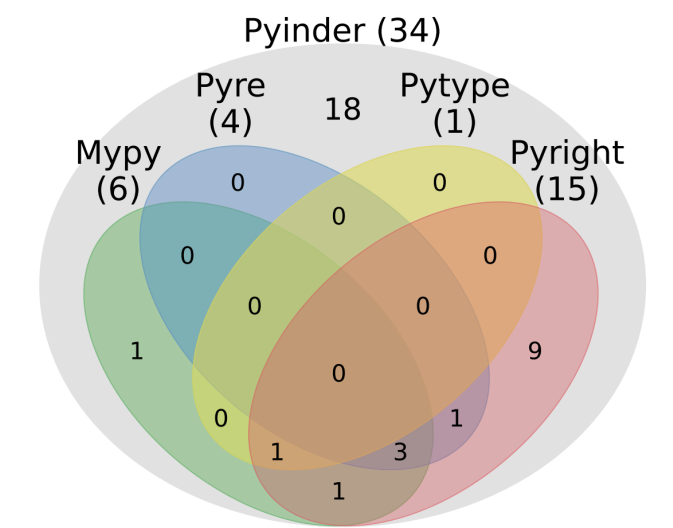


Table 2: Comparison with existing type analysis tools on the # of alarms and analysis time (both per 1KLoC).

	Mypy	Pyre	Pytype	Pyright	Pyinder
#alarms/KLoC	4.72	8.38	7.74	11.37	4.57
time(s)/KLoC	0.48	0.26	98.28	0.98	5.78

We reports **fewer alarms** than existing tools

RQ1: Effectiveness

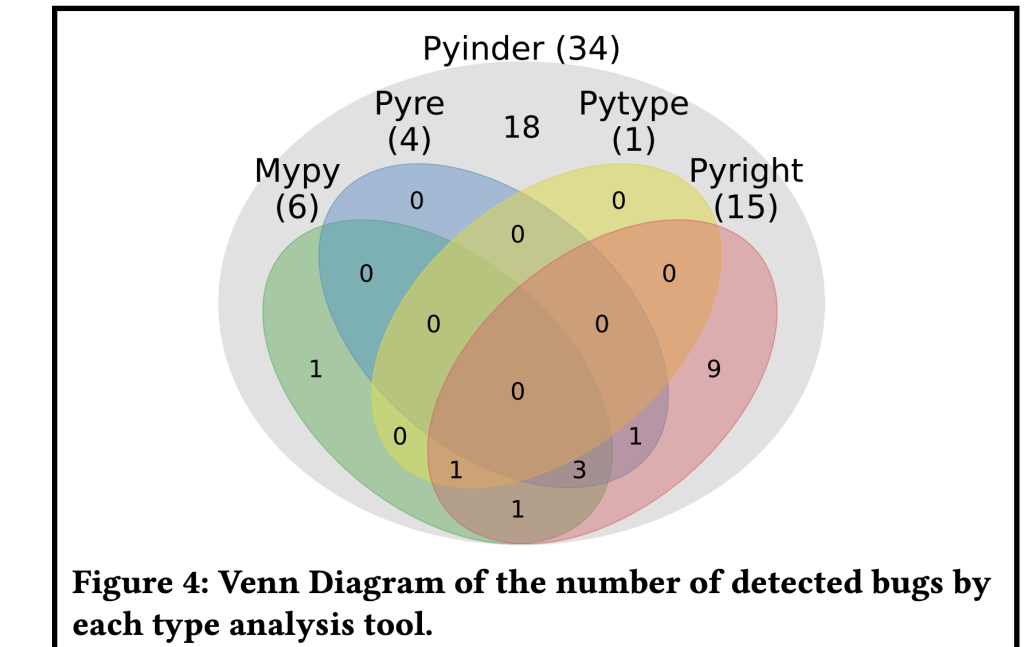


Table 2: Comparison with existing type analysis tools on the # of alarms and analysis time (both per 1KLoC).

	Mypy	Pyre	Pytype	Pyright	Pyinder
#alarms/KLoC	4.72	8.38	7.74	11.37	4.57
time(s)/KLoC	0.48	0.26	98.28	0.98	5.78

≤ 10 min
on average

We reports **fewer alarms** than existing tools

RQ2: Ablation Study

Table 3: V1 through V9 indicate the variants of Pyinder where each corresponding detailed point introduced in Table 1 is excluded from the analysis.

V1	Merging into Any at control flow merge points
V2	Merging into Any in collections
V3	Merging dictionaries in a key-type-insensitive manner
V4	Deep call depths (up to 6)
V5	Context-insensitive analysis
V6	No usage-based type inference
V7	Usage-based type inference without scoring
V8	Only considering member types in <code>__init__</code> method
V9	No filtering out unintended types

RQ2: Ablation Study

Table 3: V1 through V9 indicate the variants of Pyinder where each corresponding detailed point introduced in Table 1 is excluded from the analysis.

V1	Merging into Any at control flow merge points
V2	Merging into Any in collections
V3	Merging dictionaries in a key-type-insensitive manner
V4	Deep call depths (up to 6)
V5	Context-insensitive analysis
V6	No usage-based type inference
V7	Usage-based type inference without scoring
V8	Only considering member types in __init__ method
V9	No filtering

V means a scenario where each feature's key points are removed one by one

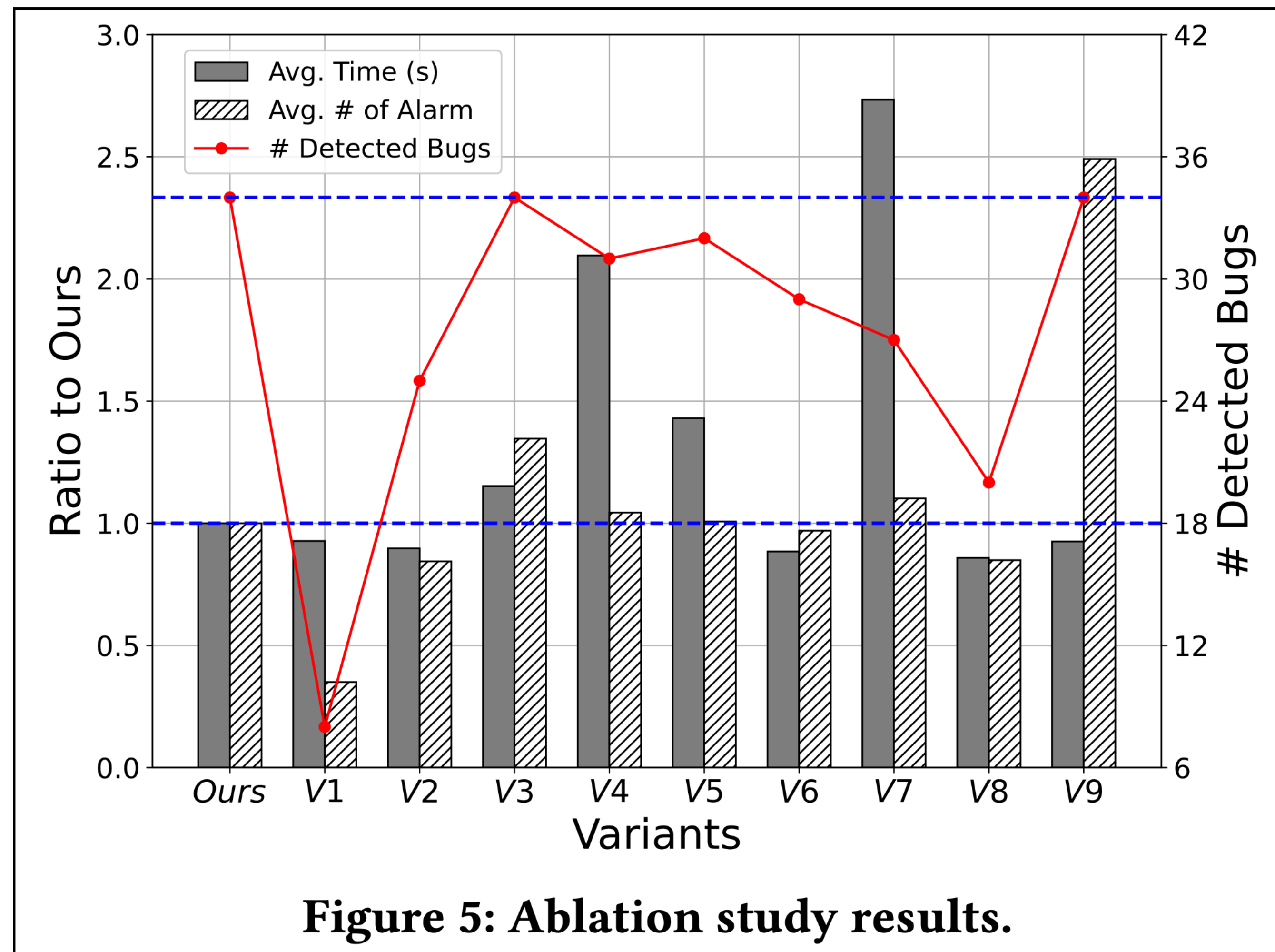
Detailed	
P1	Preserving types at control flow merge points
P2	Preserving element types in collections
P3	Considering dictionaries with a key-type-sensitive manner
P4	Shallow call depths (up to 3)
P5	Lightweight type-based context sensitivity
P6	Usage-based type inference
P7	Selecting likely types among usage-based types
P8	Considering class invariants in terms of member types
P9	Filtering out unintended member types

RQ2: Ablation Study

Table 3: V1 through V9 indicate the variants of Pyinder where each corresponding detailed point introduced in Table 1 is excluded from the analysis.

V1	Merging into Any at control flow merge points
V2	Merging into Any in collections
V3	Merging dictionaries in a key-type-insensitive manner
V4	Deep call depths (up to 6)
V5	Context-insensitive analysis
V6	No usage-based type inference
V7	Usage-based type inference without scoring
V8	Only considering member types in <code>__init__</code> method
V9	No filtering out unintended types

Detailed points for supporting the feature	
P1	Preserving types at control flow merge points
P2	Preserving element types in collections
P3	Considering dictionaries with a key-type-sensitive manner
P4	Shallow call depths (up to 3)
P5	Lightweight type-based context sensitivity
P6	Usage-based type inference
P7	Selecting likely types among usage-based types
P8	Considering class invariants in terms of member types
P9	Filtering out unintended member types



RQ2: Ablation Study

Table 3: V1 through V9 indicate the variants of Pyinder where each corresponding detailed point introduced in Table 1 is excluded from the analysis.

V1	Merging into Any at control flow merge points
V2	Merging into Any in collections
V3	Merging dictionaries in a key-type-insensitive manner
V4	Deep call depths (up to 6)
V5	Context-insensitive analysis
V6	No usage-based type inference
V7	Usage-based type inference without scoring
V8	Only considering member types in <code>__init__</code> method
V9	No filtering out unintended types

P1	
P2	
P3	Considering dictionaries with a key-type-sensitive manner
P4	Shallow call depths (up to 3)
P5	Lightweight type-based context sensitivity
P6	Usage-based type inference
P7	Selecting likely types among usage-based types
P8	Considering class invariants in terms of member types
P9	Filtering out unintended member types

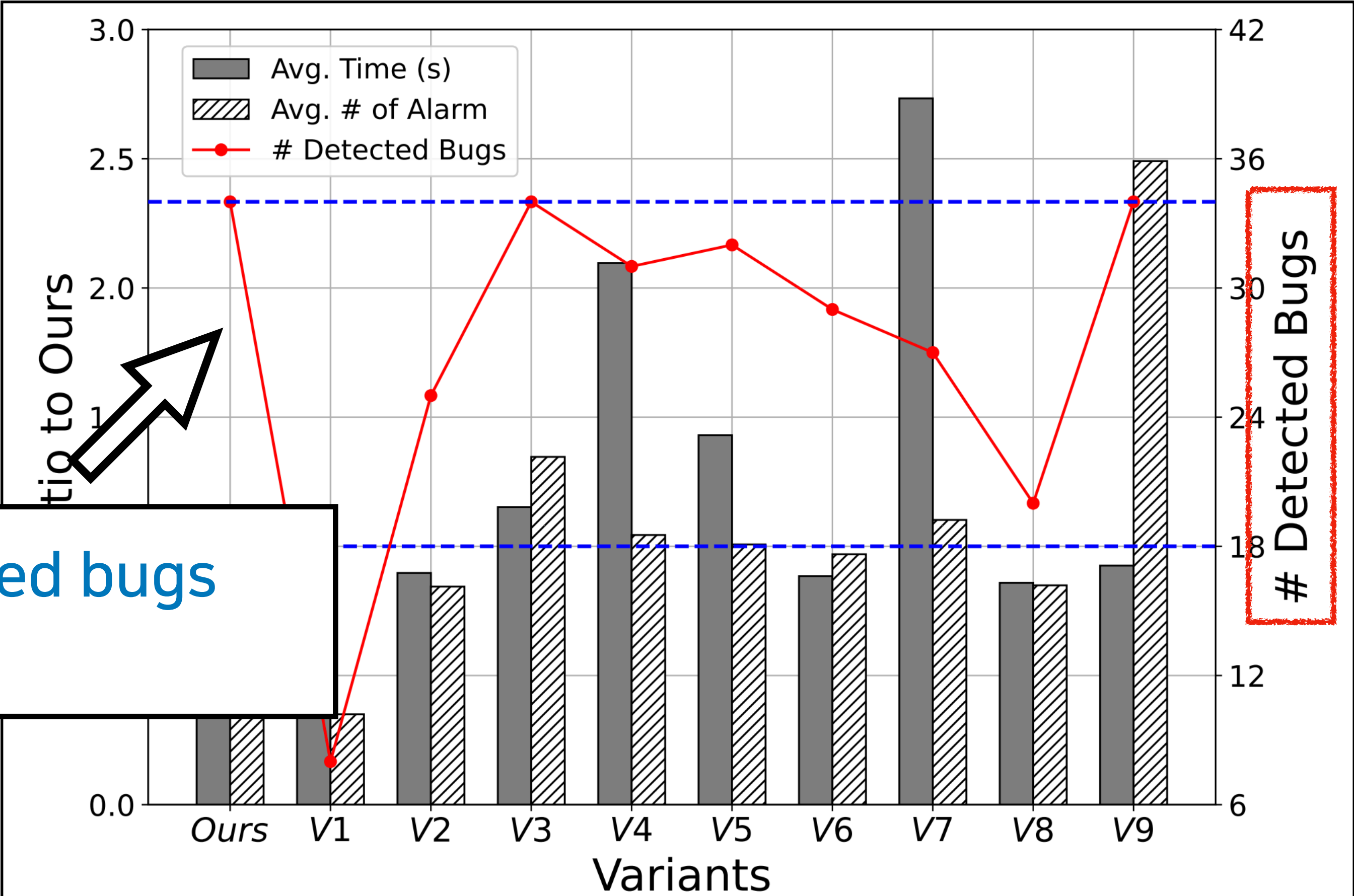


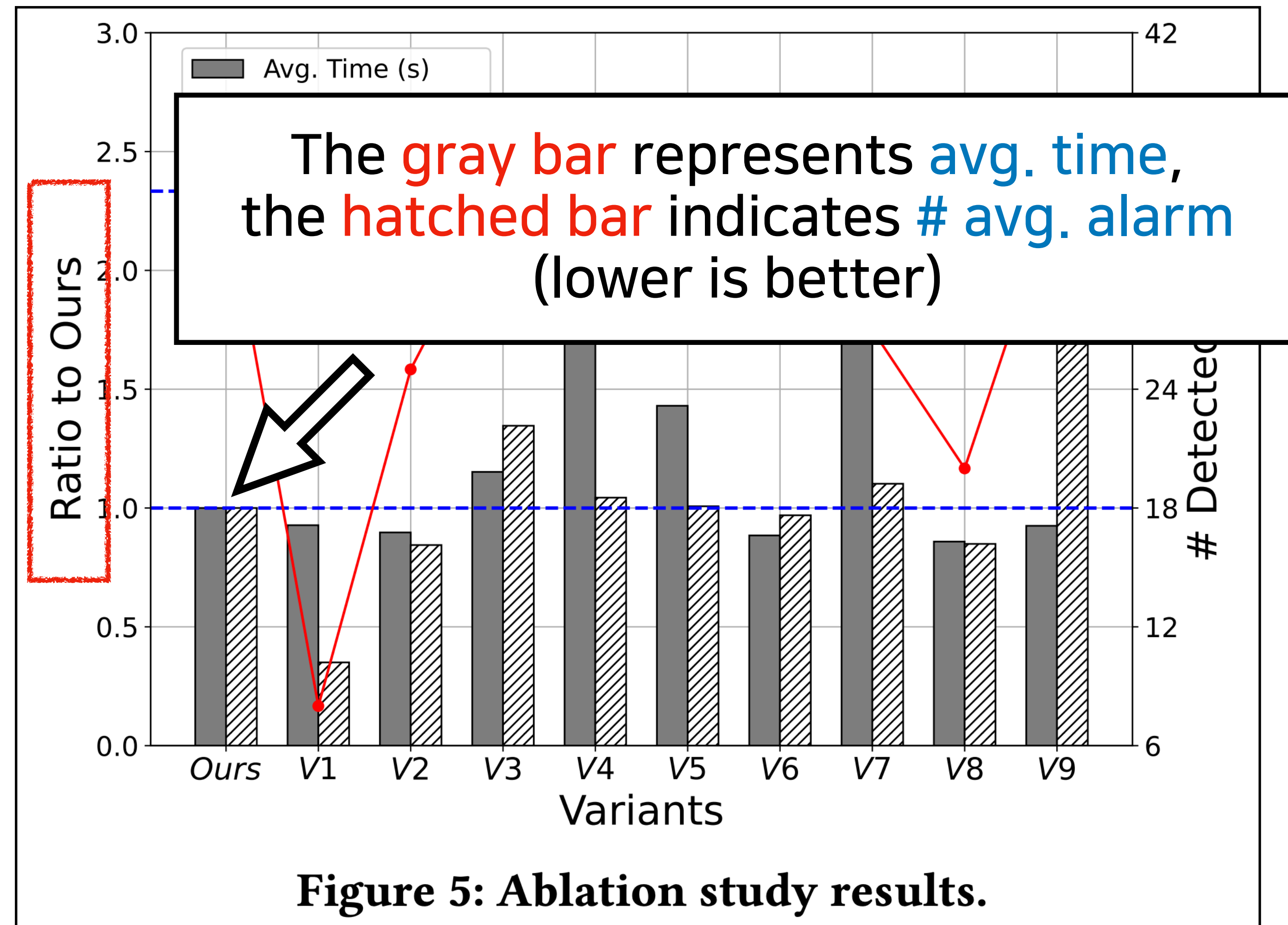
Figure 5: Ablation study results.

RQ2: Ablation Study

Table 3: V1 through V9 indicate the variants of Pyinder where each corresponding detailed point introduced in Table 1 is excluded from the analysis.

V1	Merging into Any at control flow merge points
V2	Merging into Any in collections
V3	Merging dictionaries in a key-type-insensitive manner
V4	Deep call depths (up to 6)
V5	Context-insensitive analysis
V6	No usage-based type inference
V7	Usage-based type inference without scoring
V8	Only considering member types in <code>__init__</code> method
V9	No filtering out unintended types

Detailed points for supporting the feature	
P1	Preserving types at control flow merge points
P2	Preserving element types in collections
P3	Considering dictionaries with a key-type-sensitive manner
P4	Shallow call depths (up to 3)
P5	Lightweight type-based context sensitivity
P6	Usage-based type inference
P7	Selecting likely types among usage-based types
P8	Considering class invariants in terms of member types
P9	Filtering out unintended member types



RQ2: Ablation Study

Table 3: V1 through V9 indicate the variants of Pyinder where each corresponding detailed point introduced in Table 1 is excluded from the analysis.

V1	Merging into Any at control flow merge points
V2	Merging into Any in collections
V3	Merging dictionaries in a key-type-insensitive manner
V4	Deep call depths (up to 6)
V5	
V6	
V7	
V8	
V9	

Compared to ours,
V4 detected 5 fewer bugs
took 2.1x more time
generated 1.1x more alarms

P1	
P2	Preserving element types in collections
P3	Considering dictionaries with a key-type-sensitive manner
P4	Shallow call depths (up to 3)
P5	Lightweight type-based context sensitivity
P6	Usage-based type inference
P7	Selecting likely types among usage-based types
P8	Considering class invariants in terms of member types
P9	Filtering out unintended member types

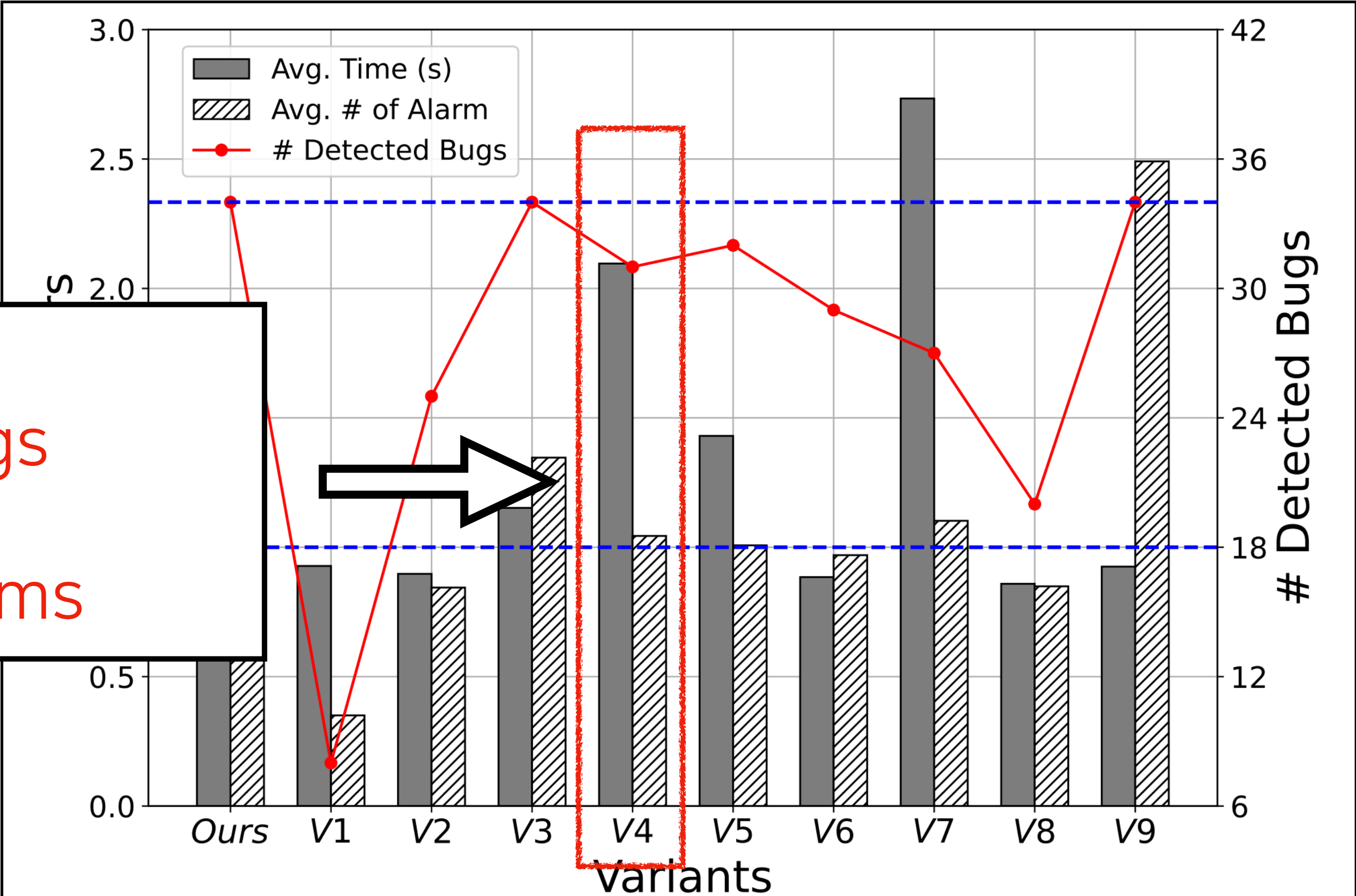


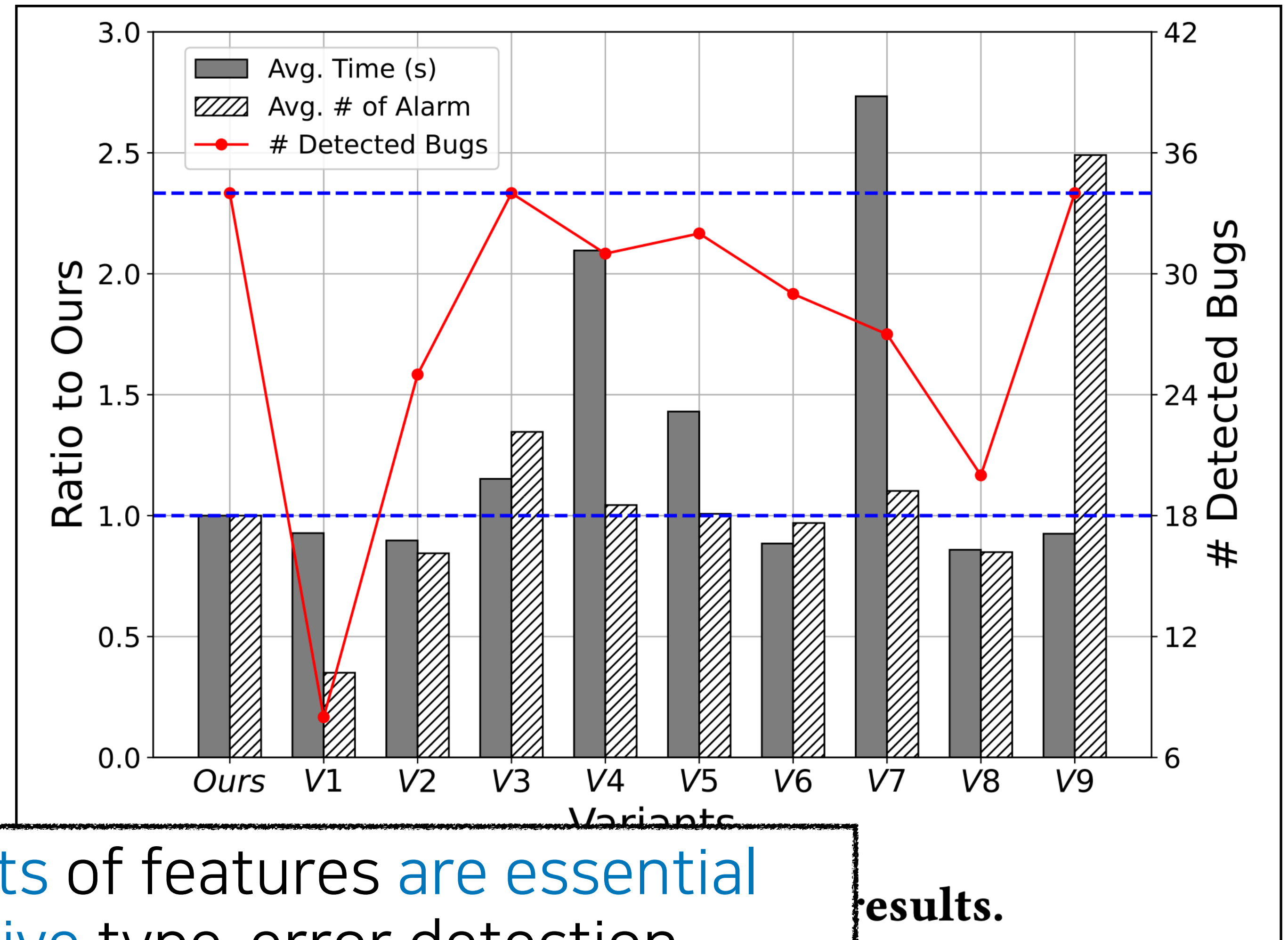
Figure 5: Ablation study results.

RQ2: Ablation Study

Table 3: V1 through V9 indicate the variants of Pyinder where each corresponding detailed point introduced in Table 1 is excluded from the analysis.

V1	Merging into Any at control flow merge points
V2	Merging into Any in collections
V3	Merging dictionaries in a key-type-insensitive manner
V4	Deep call depths (up to 6)
V5	Context-insensitive analysis
V6	No usage-based type inference
V7	Usage-based type inference without scoring
V8	Only considering member types in <code>__init__</code> method
V9	No filtering out unintended types

Detailed points for supporting the feature	
P1	Preserving types at control flow merge points
P2	Preserving element types in collections
P3	Considering dictionaries with a key-type-sensitive manner
P4	Shallow call depths (up to 3)
P5	Lightweight type-based context sensitivity
P6	Usage-based type inference
P7	Selecting likely types among usage-based types
P8	Considering class invar
P9	Filtering out unintended



All detailed points of features are essential for the effective type-error detection

results.

RQ3: Generality

Table 4: Detected bugs in recent open-source Python projects. Status indicates whether the bug has been confirmed or not. (○ : Confirmed/Fixed, × : Won't Fix, △ : Duplicated, - : No response.)

Bug	Status	Mypy	Pyre	Pyright	Pyinder
core-1	-	×	×	✓	✓
kivy-1	-	✓	✓	✓	✓
kivy-2	○	✓	×	✓	✓
kivy-3	-	✓	✓	✓	✓
kivy-4	×	✓	✓	✓	✓
kivy-5	-	×	×	✓	✓
langchain-1	○	×	×	×	✓
luigi-1	△	×	×	×	✓
luigi-2	-	✓	✓	✓	✓
pwntools-1	○	✓	✓	✓	✓
pwntools-2	○	×	×	×	✓
pwntools-3	○	×	✓	✓	✓
pwntools-4	○	×	✓	✓	✓
telegram*-1	○	×	×	×	✓
scipy-1	○	✓	✓	✓	✓
scipy-2	○	✓	✓	✓	✓
scipy-3	△	✓	✓	✓	✓
scrapy-1	○	×	×	×	✓
tqdm-1	-	×	×	×	✓

*python-telegram-bot

RQ3: Generality

Table 4: Detected bugs in recent open-source Python projects. Status indicates whether the bug has been confirmed or not. (○ : Confirmed/Fixed, × : Won't Fix, △ : Duplicated, - : No response.)

Bug	Status	Mypy	Pyre	Pyright	Pyinder
core-1	-	×	×	✓	✓
kivy-1	-	✓	✓	✓	✓
kivy-2	○	✓	×	✓	✓
kivy-3	-	✓	✓	✓	✓
kivy-4	×	✓	✓	✓	✓
kivy-5	-	×	×	✓	✓
langchain-1	○	×	×	×	✓
luigi-1	△	×	×	×	✓
luigi-2	-	✓	✓	✓	✓
pwntools-1	○	✓	✓	✓	✓
pwntools-2	○	×	×	×	✓
pwntools-3	○	×	✓	✓	✓
pwntools-4	○	×	✓	✓	✓
telegram*-1	○	×	×	×	✓

We successfully found 19 previously-unknown type errors with five times fewer alarms compared to Pyright

Limitations & Future directions

- False Negative (34 type errors)
 - 29 cases were not detected due to unknown **external libraries and inputs**

```
1 import ext
2 # ext is an external lib
3 # ext.f returns a string
4 x = ext.f()
5 x+1 # TypeError: str+int
```

(a) Example that requires the return type of external function `ext.f()`.

```
1 # conf.json : {'a': 1}
2 f = open("conf.json")
3 conf_dict = json.loads(f)
4 # TypeError: dict + list
5 conf_dict + [1]
```

(b) Example that requires the content of the external input `conf.json`.

Limitations & Future directions

- False Negative (34 type errors)
- 29 cases were not detected due to unknown **external libraries and inputs**

```
1 import ext
2 # ext is an external lib
3 # ext.f returns a string
4 x = ext.f()
5 x+1 # TypeError: str+int
```

(a) Example that requires the

return
ext.f() returns **str type**
but it comes from
unknown external library

```
1 # conf.json : {'a': 1}
2 f = open("conf.json")
3 conf_dict = json.loads(f)
4 # TypeError: dict + list
5 conf_dict + [1]
```

(b) Example that requires the
content of the external input
conf.json.

Limitations & Future directions

- False Negative (34 type errors)
 - 29 cases were not detected due to unknown **external libraries and inputs**

```
1 import ext
2 # ext is an external lib
3 # ext.f returns a string
4 x = ext.f()
5 x+1 # TypeError: str+int
```

(a) Example that requires the

return
tion
ext.f() returns **str type**
but it comes from
unknown external library

```
1 # conf.json : {'a': 1}
2 f = open("conf.json")
3 conf_dict = json.loads(f)
4 # TypeError: dict + list
5 conf_dict + [1]
```

(b) Example that requires the

conf
conf
conf_dict is **dictionary**
but it comes from
unknown external input

Limitations & Future directions

- False Negative (34 type errors)
- 29 cases were not detected due to

We need an **effective way**
to **infer types of external libraries and inputs**
while **minimizing negative effects** (e.g., the analysis cost)

```
4 x = ext.f()  
5 x+1 # TypeError: str+int
```

(a) Example that requires the

return
tion
ext.f() returns **str type**
but it comes from
unknown external library

```
4 # TypeError: dict + list  
5 conf_dict + [1]
```

(b) Example that requires the

conf
conf
conf_dict is **dictionary**
but it comes from
unknown external input

Limitations & Future directions

- False Negative (34 type errors)
 - 29 cases were not detected due to unknown **external libraries and inputs**
 - 5 cases require **analysis other** than type analysis (e.g., value analysis)

Limitations & Future directions

- False Positive
- We have conducted a limited investigation of false positives
 - too many alarms (37,285)
 - hard to determine whether it is true or false

Limitations & Future directions

- False Positive
- We have conducted a limited investigation of false positives
 - too many alarms (37,285)
 - hard to determine whether it is true or false
- We sample 269 alarms for the investigation (confidence interval: 90% and margin of error: 5%)
 - We found 106 false alarms (others are difficult to determine...)

Limitations & Future directions

- False Positive
 - We found two representatives cases that account for 70% of the false alarms

```
1 class Symbol:
2     def __radd__(
3         self, x
4     ): return x+1
5 # False Alarm
6 # int.__add__(1, Symbol)
7 1+Symbol()
```

(a) False alarm at line 7 due to the limitation of modeling magic methods.

```
1 def f(x, y):
2     # False Alarm
3     if x > 0: y+1
4     # False Alarm
5     else:      y+'1'
6 f(1, 1)
7 f(-1, '1')
```

(b) False alarm at lines 3 and 5 due to the limitation of type analysis.

Limitations & Future directions

- False Positive
 - We found two representatives cases that account for 70% of the false alarms

```
1 class Symbol:
2     def __radd__(
3         self, x
4     ): return x+1
5 # False Alarm
6 # int.__add__
7 1+Symbol()
```

Analyzer translates...

(a) False
to the lin
magic methods.

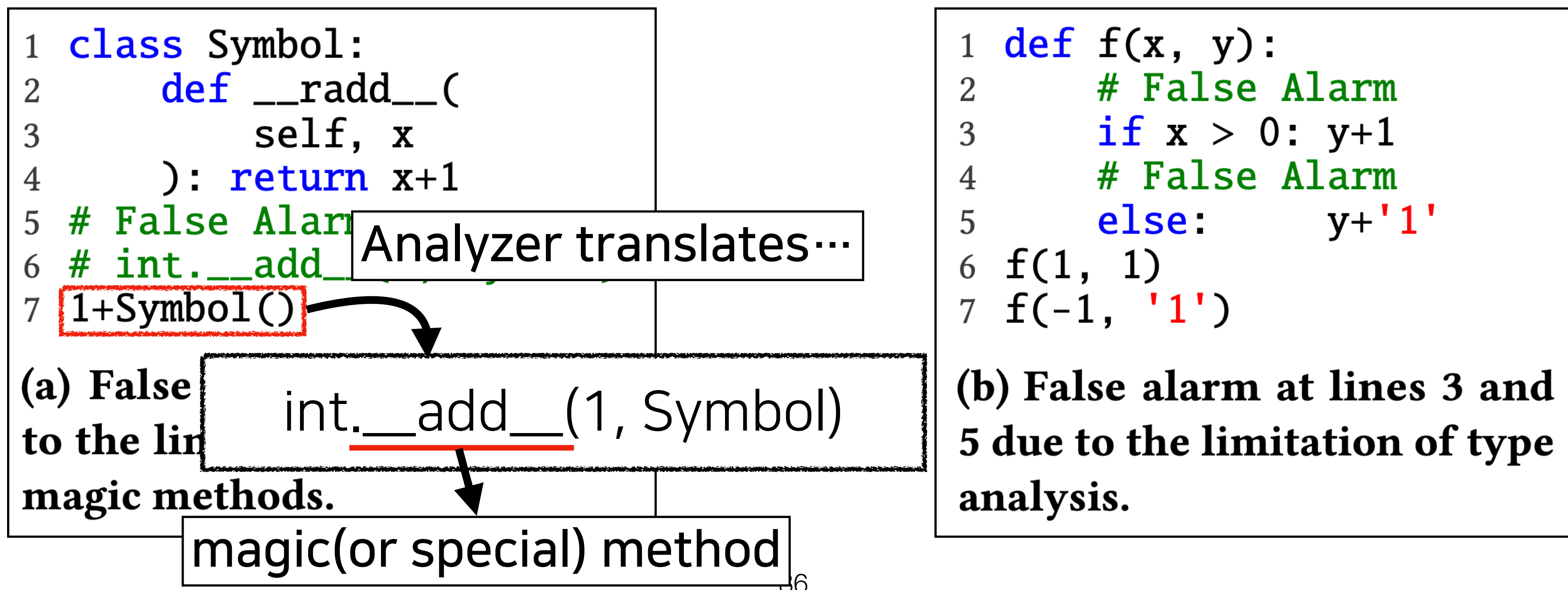
int.__add__(1, Symbol)

```
1 def f(x, y):
2     # False Alarm
3     if x > 0: y+1
4     # False Alarm
5     else:      y+'1'
6 f(1, 1)
7 f(-1, '1')
```

(b) False alarm at lines 3 and
5 due to the limitation of type
analysis.

Limitations & Future directions

- False Positive
 - We found two representatives cases that account for 70% of the false alarms



Limitations & Future directions

- False Positive
- We found two representative cases that account for 70% of the false alarms

```
1 class Symbol:
2     def __radd__(
3         self, x
4     ): return x+1
5 # False Alarm
6 # int.__add__(1, Symbol)
7 1+Symbol()
```

(a) False
to the lin
magic methods.

int.__add__(1, Symbol)

No `__add__` method
in class Symbol
=> **TypeError**

```
def f(x, y):
# False Alarm
if x > 0: y+1
# False Alarm
else:      y+'1'
6 f(1, 1)
7 f(-1, '1')
```

(b) False alarm at lines 3 and
5 due to the limitation of type
analysis.

Limitations & Future directions

- False Positive
- We found two representative cases that account for 70% of the false alarms

```
1 class Symbol:
2     def __radd__(
3         self, x
4     ): return x+1
5 # False Alarm
6 # int.__add__(1, Symbol)
7 1+Symbol()
```

(a) False alarm to the line 7 due to the limitation of type analysis of magic methods.

No `__add__` method in class `Symbol` => `TypeError`

```
def f(x, y):
# False Alarm
if x > 0: y+1
# False Alarm
else: y+'1'
6 f(1, 1)
7 f(-1, '1')
```

(b) False alarm at lines 3 and 5 due to the limitation of type analysis.

Limitations & Future directions

- False Positive
- We found two representatives cases that account for 70% of the false alarms

```
1 class Symbol:
2     def __radd__(
3         self, x
4     ): return x+1
5 # False Alarm
6 # int.__add__(1, Symbol)
7 1+Symbol()
```

(a) False alarm due to the limitation of type analysis of magic methods.

Symbol.__radd__(Symbol, 1)

Actual translation

No `__add__` method in class `Symbol` => `TypeError`

```
1 def f(x, y):
2     # False Alarm
3     if x > 0: y+1
4     # False Alarm
5     else: y+'1'
6 f(1, 1)
7 f(-1, '1')
```

(b) False alarm at lines 3 and 5 due to the limitation of type analysis.

Limitations & Future directions

- False Positive
- We find a false alarm
- `__radd__`

These methods are called to implement the binary arithmetic operations (+, -, *, @, /, //, %, [divmod\(\)](#), [pow\(\)](#), **, <<, >>, &, ^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation [3] and the operands are of different types. [4] For instance, to evaluate the expression `x - y`, where `y` is an instance of a class that has an [__rsub__\(\)](#) method, `type(y).__rsub__(y, x)` is called if `type(x).__sub__(x, y)` returns [NotImplemented](#).

Note that ternary [pow\(\)](#) will not try calling [__rpow__\(\)](#) (the coercion rules would become too complicated).

```
1 class Symbol:
2
3
4
5 # ...
6 # ...
7 1 + Symbol('x')
```

Note: If the right operand's type is a subclass of the left operand's type and that subclass provides a different implementation of the reflected method for the operation, this method will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors' operations.

(a) False alarm
to the left
magic methods.

`Symbol.__radd__(Symbol, 1)`

Actual translation

(b) False alarm at lines 3 and 5 due to the limitation of type analysis.

Limitations & Future directions

- False Positive
 - We found two representatives cases that account for 70% of the false alarms

```
1 class Symbol:
2     def __radd__(
3         self, x
4     ): return x+1
5 # False Alarm
6 # int.__add__(1, Symbol)
7 1+Symbol()
```

(a) False alarm at line 7 due to the limitation of modeling magic methods.

```
1 def f(x, y):
2     # False Alarm
3     if x > 0: y+1
4     # False Alarm
5     else: y+'1'
6     f(1, 1)
7     f(-1, '1')
```

(b) False alarm at lines 3 and 5

False alarms occur because type domain cannot determine the branches

Lessons

- Current type analysis tools have limitations in detecting real-world type errors
- Our features are novel for effective type-error detection
- The necessity of handling tricky magic method modeling
- There is room for further advancement in detecting type errors or pruning false alarms

예상되는 질문

- 왜 sound하게 고정한 후 줄이는 형태로 가지 않았느냐?
- 68개의 대한 예제에 대해 핏 된거 아니냐?