



Return of CFA: Call-Site Sensitivity Can Be Superior to Object Sensitivity Even for Object-Oriented Programs

Minseok Jeon and Hakjoo Oh



POPL 2022 @ Philadelphia, USA



Two major camps

A:
Call-Site Sensitivity
Object Sensitivity

Can
Even for
Object-Oriented Programs

Minseok Jeon and Hakjoo Oh



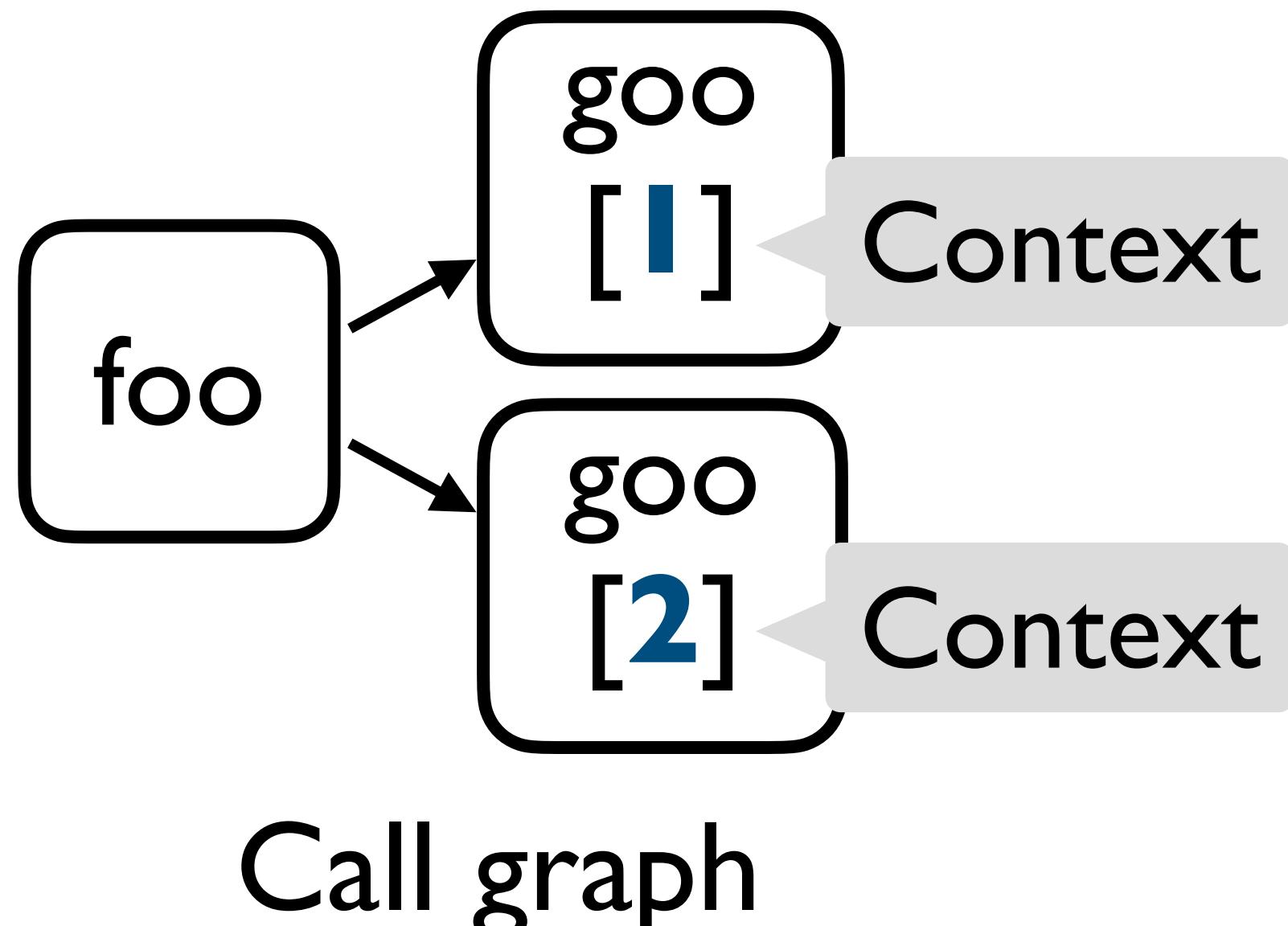
POPL 2022 @ Philadelphia, USA

Call-site Sensitivity vs Object Sensitivity

Call-site sensitivity was born in 1981

- Considers “**Where**”

```
0: foo(){  
1:   goo();  
2:   goo();  
3: }
```



1981

2002

2010

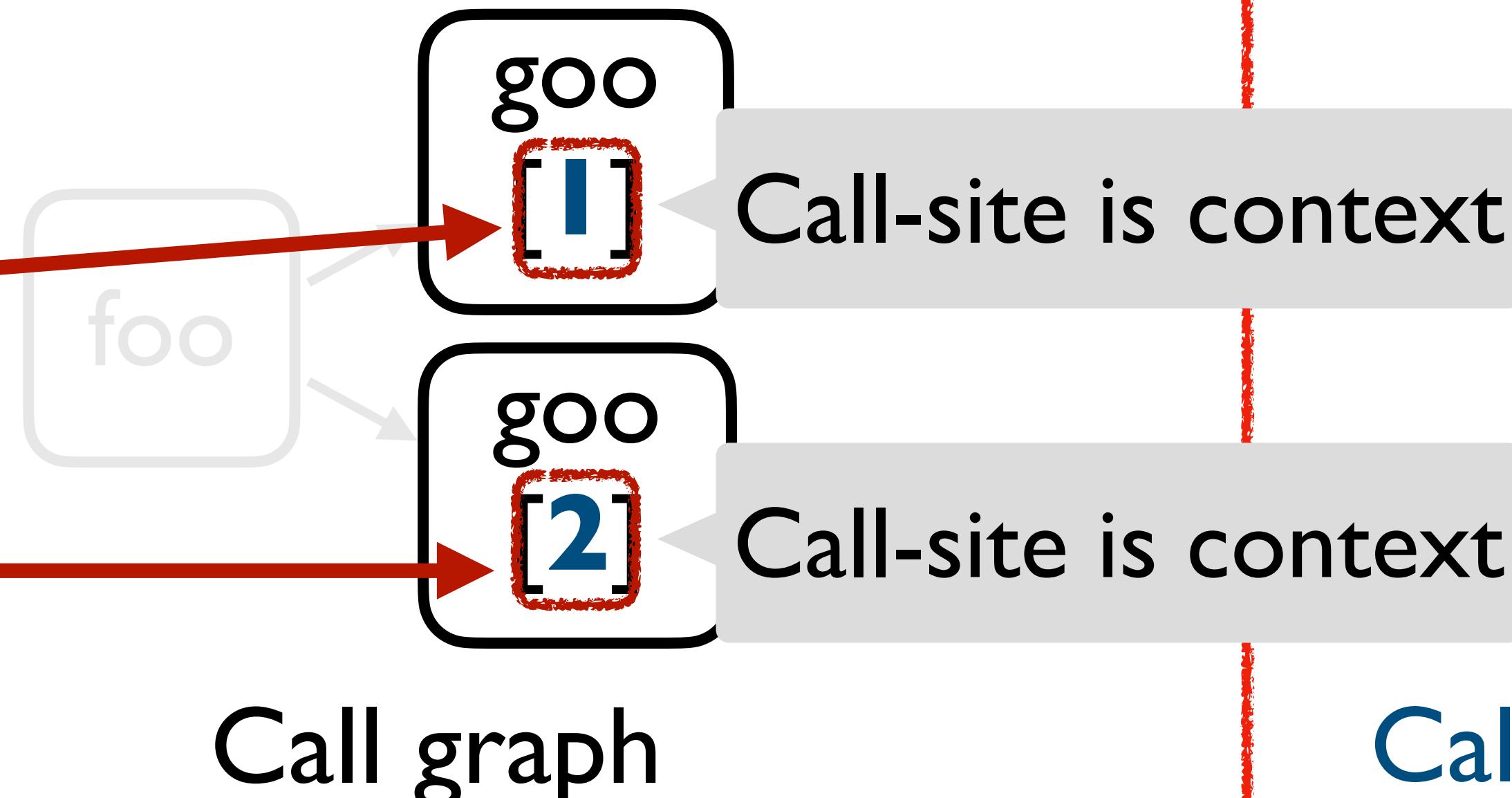
2022

Call-site Sensitivity vs Object Sensitivity

Call-site sensitivity was born in 1981

- Considers “**Where**”

```
0: foo(){  
    1:     goo();  
    2:     goo();  
    3: }
```



Where is it called from?

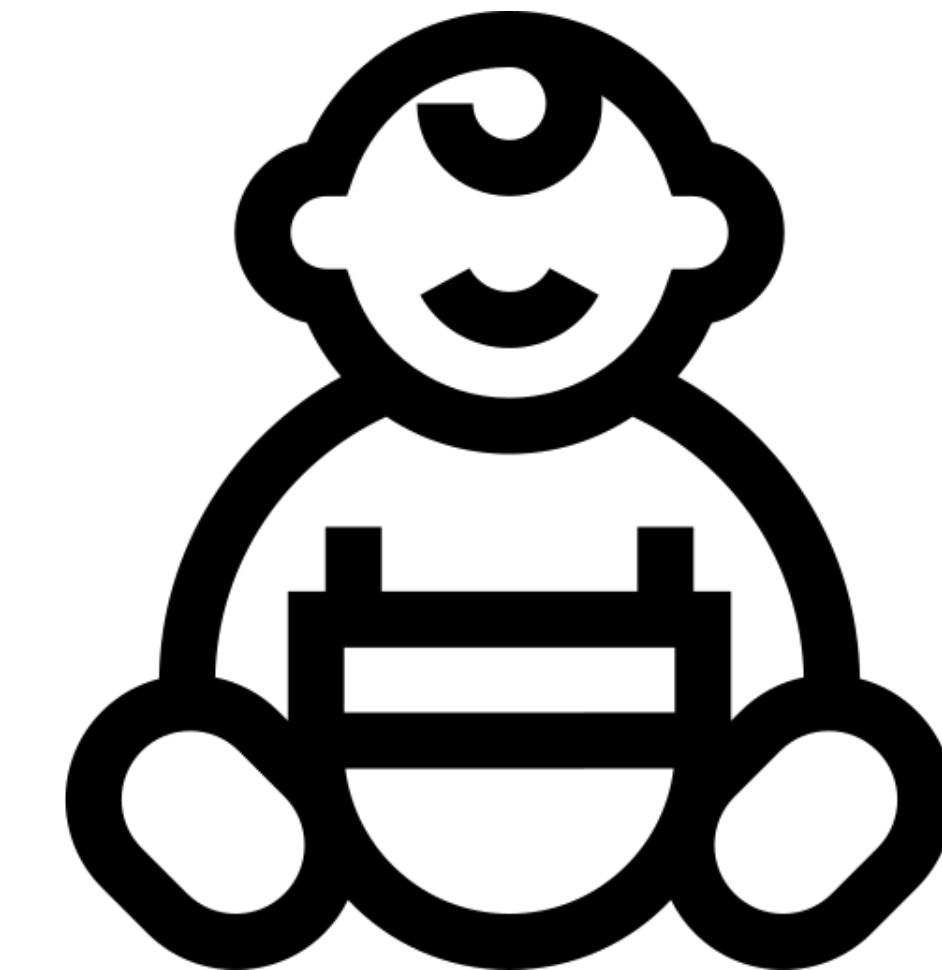
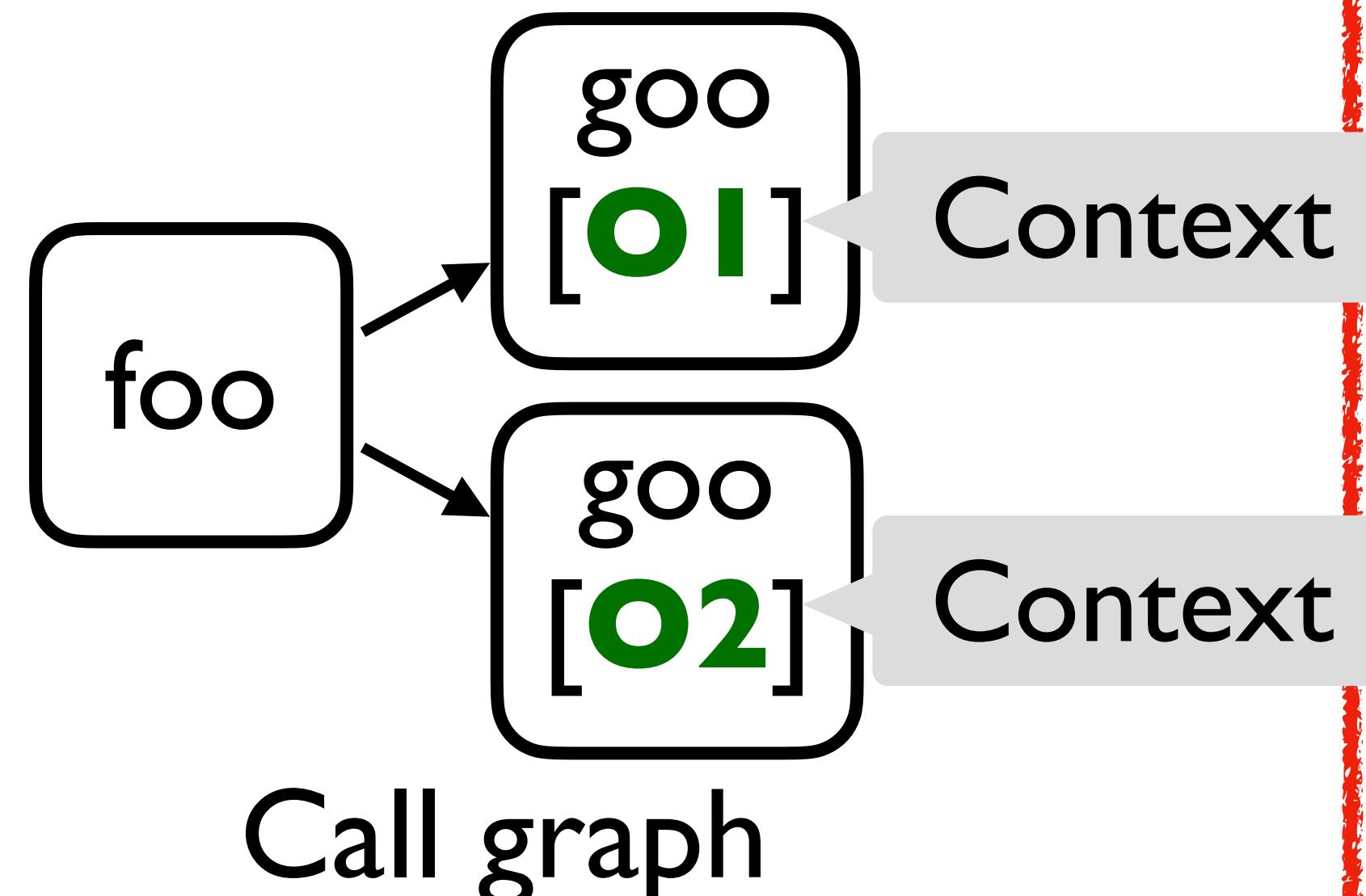
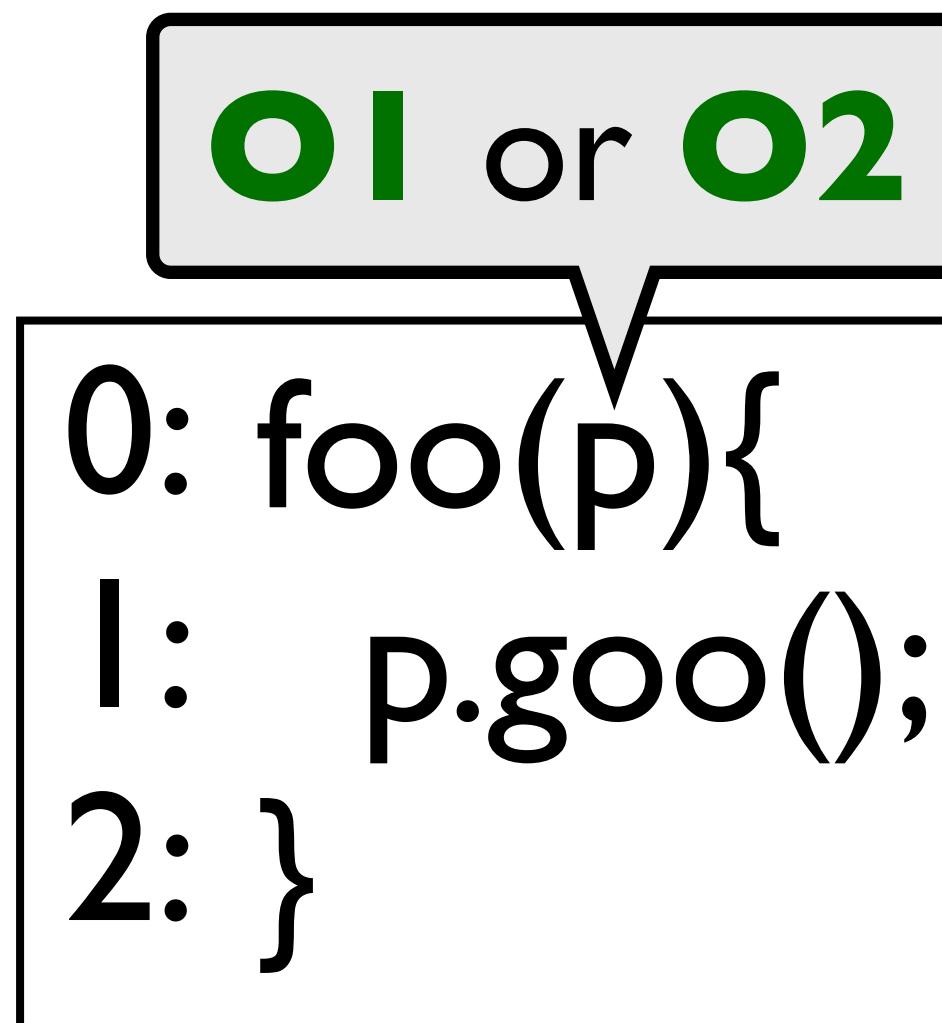


Call-site sensitivity

Call-site Sensitivity vs Object Sensitivity

Object sensitivity appeared in 2002

- Considers “**What**”



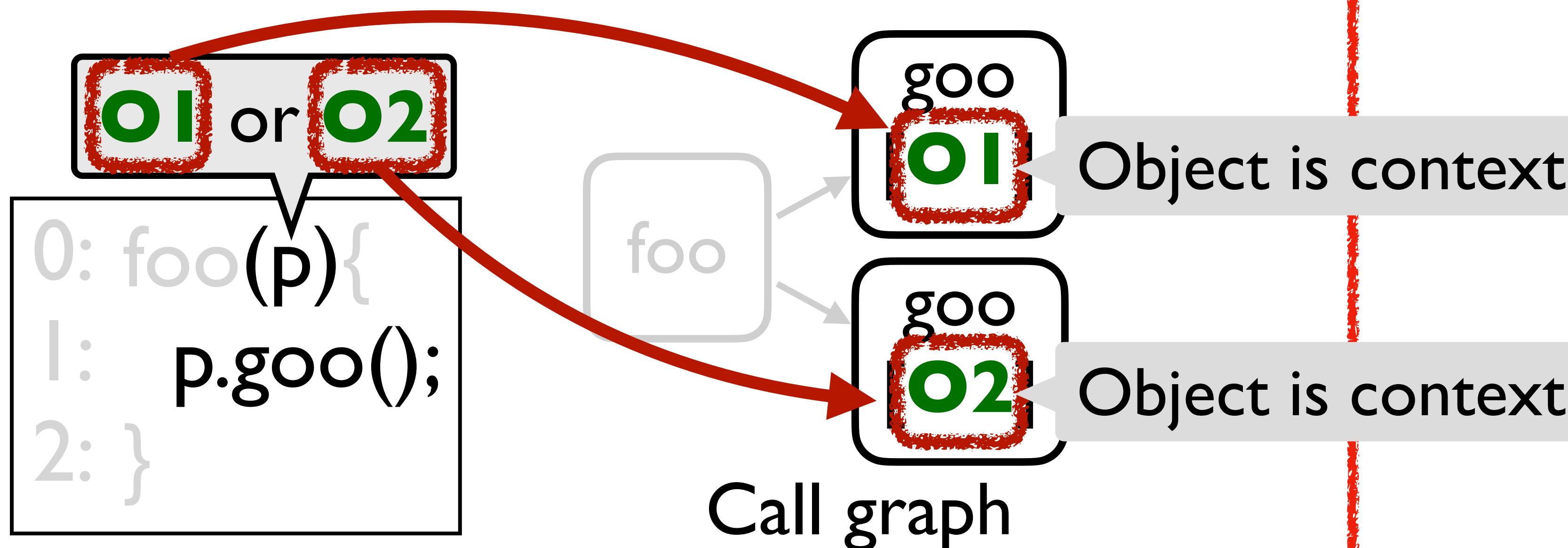
Object sensitivity



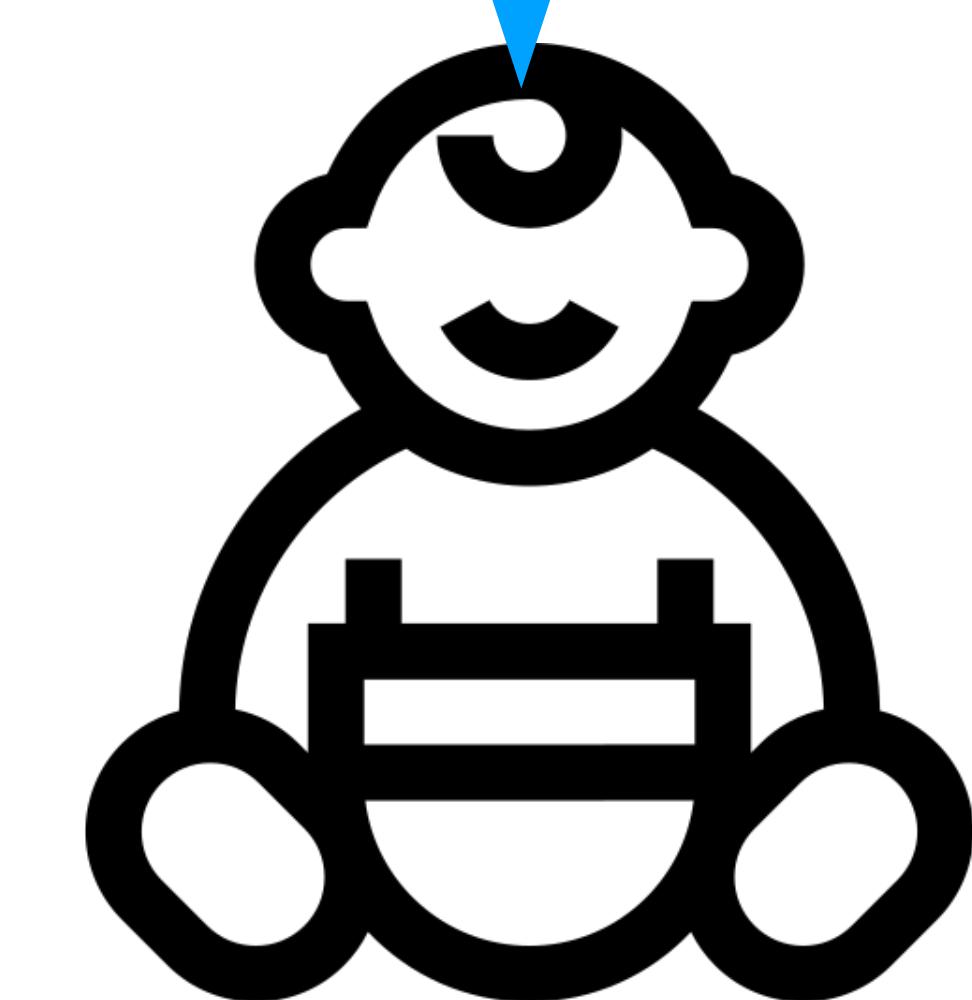
Call-site Sensitivity vs Object Sensitivity

Object sensitivity appeared in 2002

- Considers “**What**”



What is it called with?



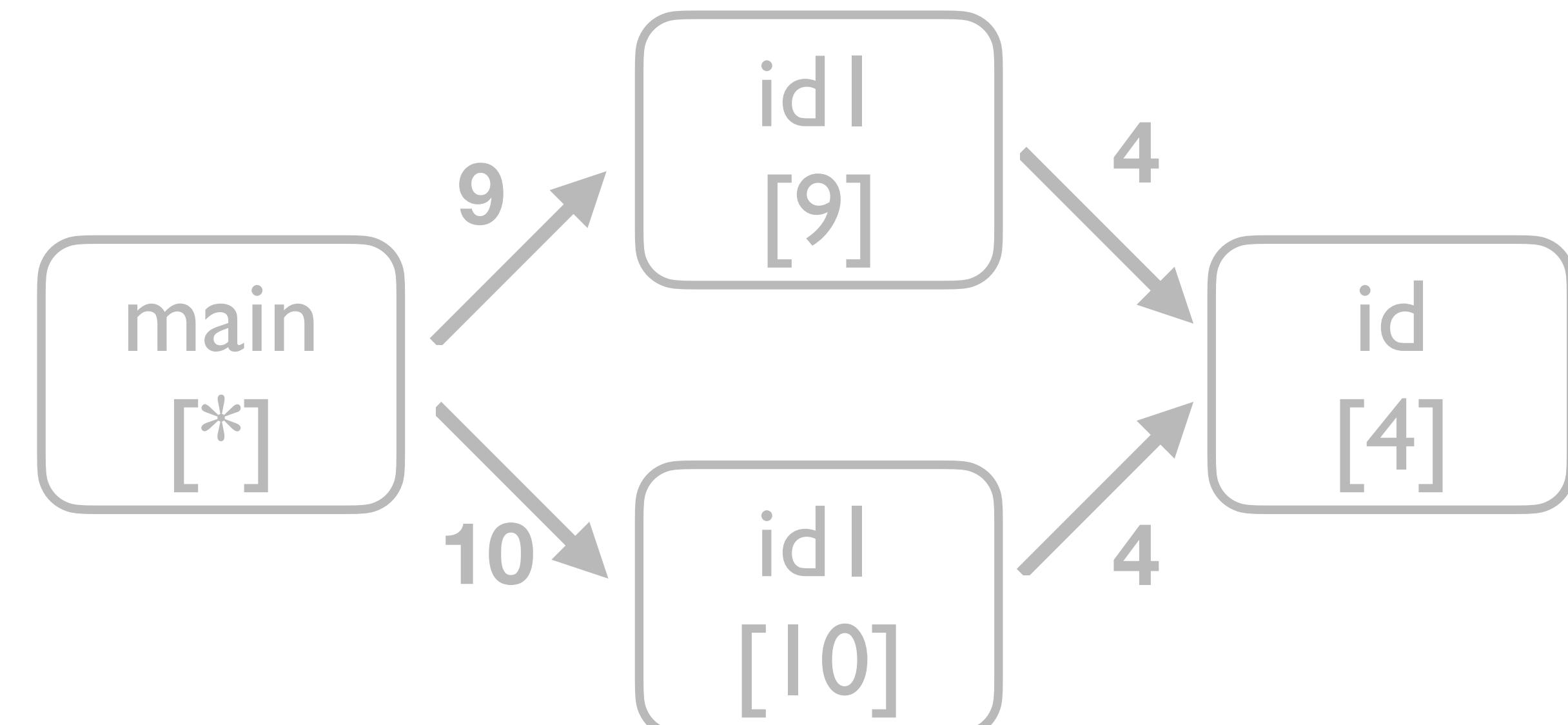
Object sensitivity



Call-site Sensitivity vs Object Sensitivity

- An example shows the **limitation** of CFA and **strength** of object sensitivity

```
0: class C{  
1:   id(v){  
2:     return v;  
3:   id1(v){  
4:     return this.id(v);}  
5: }  
6: main(){  
7:   c1 = new C();//C1  
8:   c2 = new C();//C2  
9:   a = (A) c1.id1(new A());//query1  
10:  b = (B) c2.id1(new B());//query2  
11: }
```

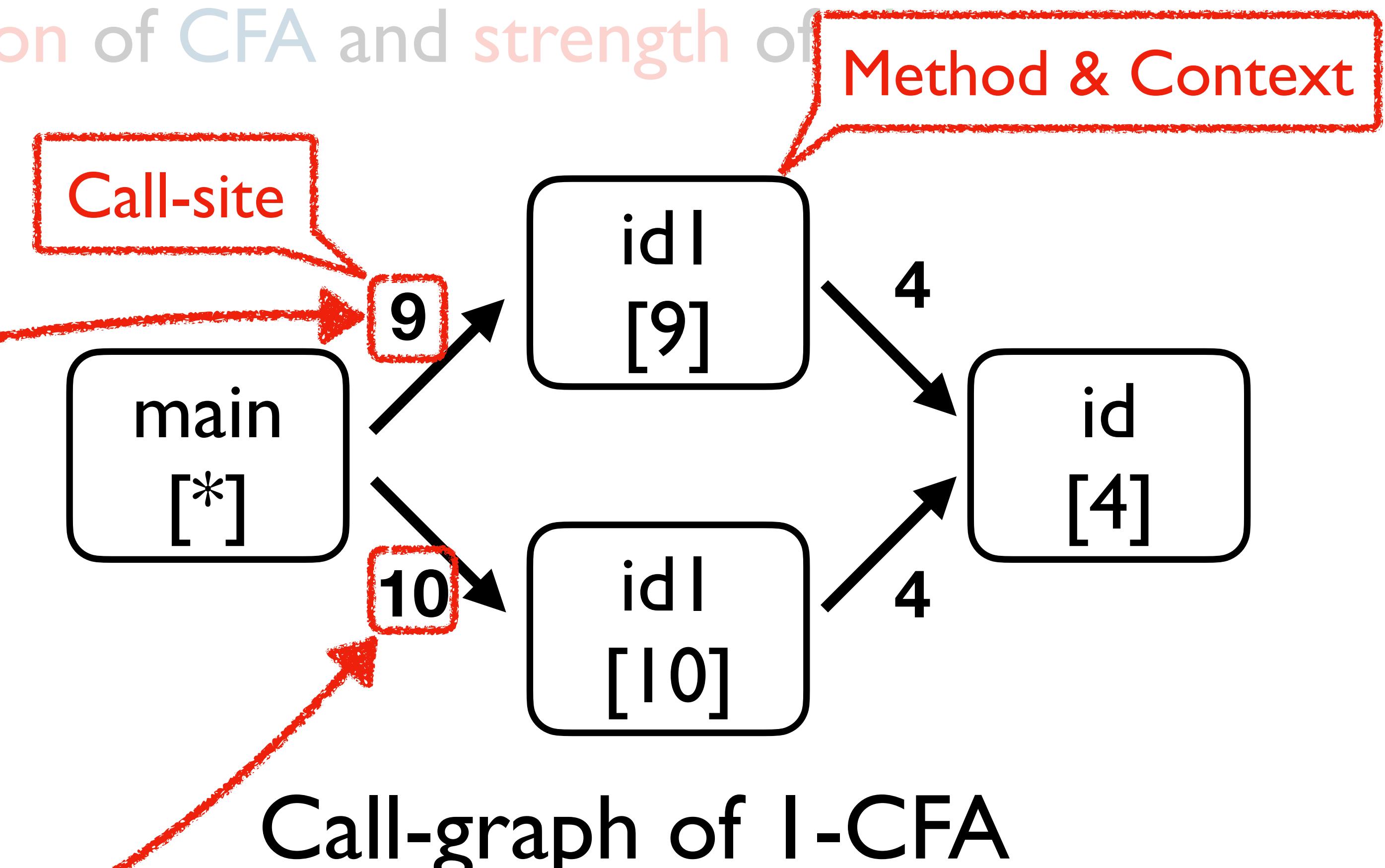


Call-graph of I-CFA

Call-site Sensitivity vs Object Sensitivity

- An example shows the **limitation** of CFA and **strength** of I-CFA

```
0: class C{  
1:   id(v){  
2:     return v;  
3:   }  
4:   id1(v){  
5:     return this.id(v);}  
6: }  
7: main(){  
8:   c1 = new C(); //C1  
9:   a = (A) c1.id1(new A()); //query1  
10:  b = (B) c2.id1(new B()); //query2  
11: }
```

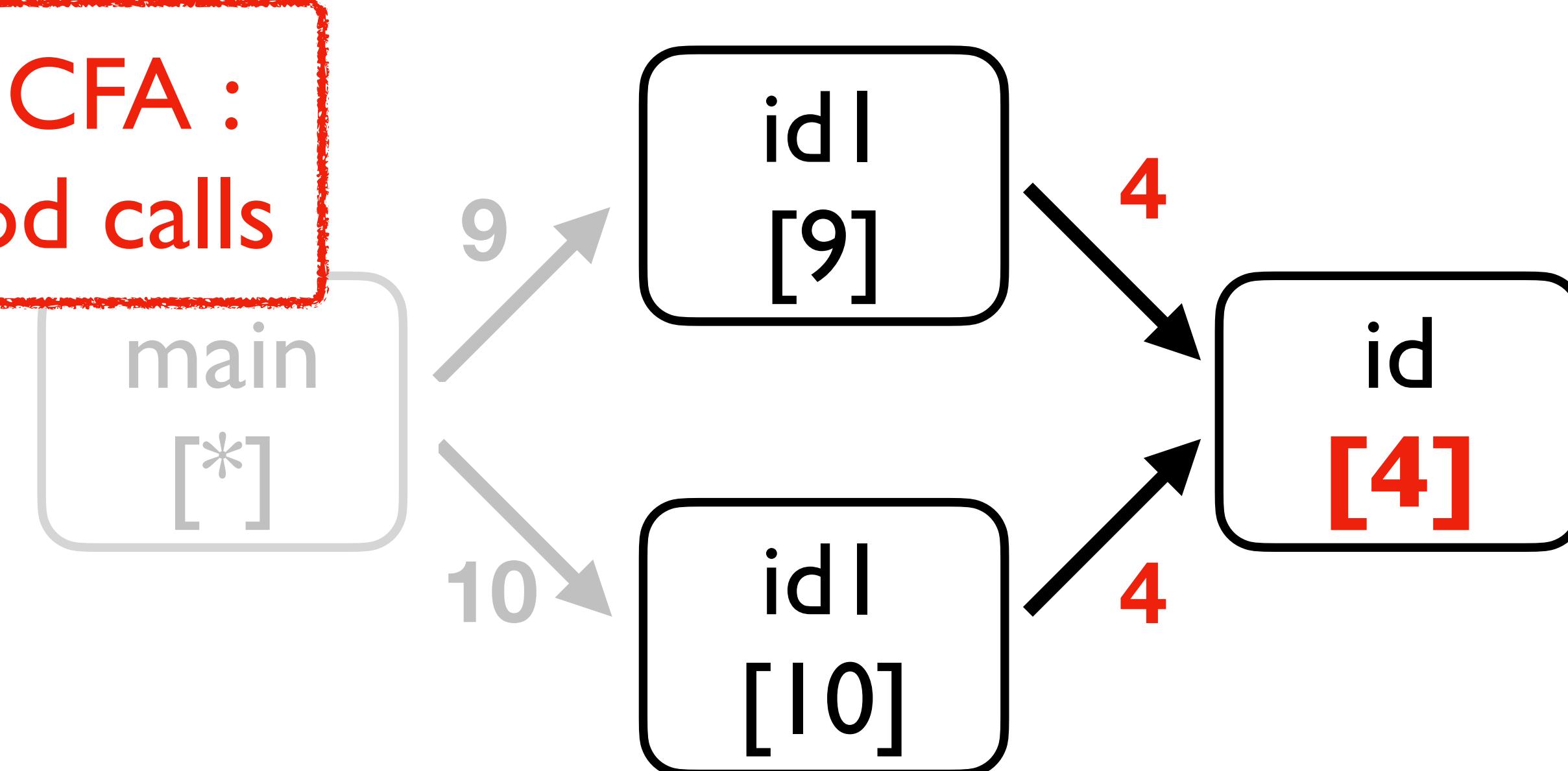


Call-site Sensitivity vs Object Sensitivity

- An example shows the **limitation** of CFA and strength of **object sensitivity**

```
0: class C{  
1:   id(v){  
2:     return v;  
3:   id1(v){  
4:     return this.id(v);}  
5: }  
6: main(){  
7: c1 = new C(); //C1  
8: c2 = new C(); //C2  
9: a = (A) c1.id1(new A()); //query  
10: b = (B) c2.id1(new B()); //query?  
11: }
```

Limitation of CFA:
Nested method calls

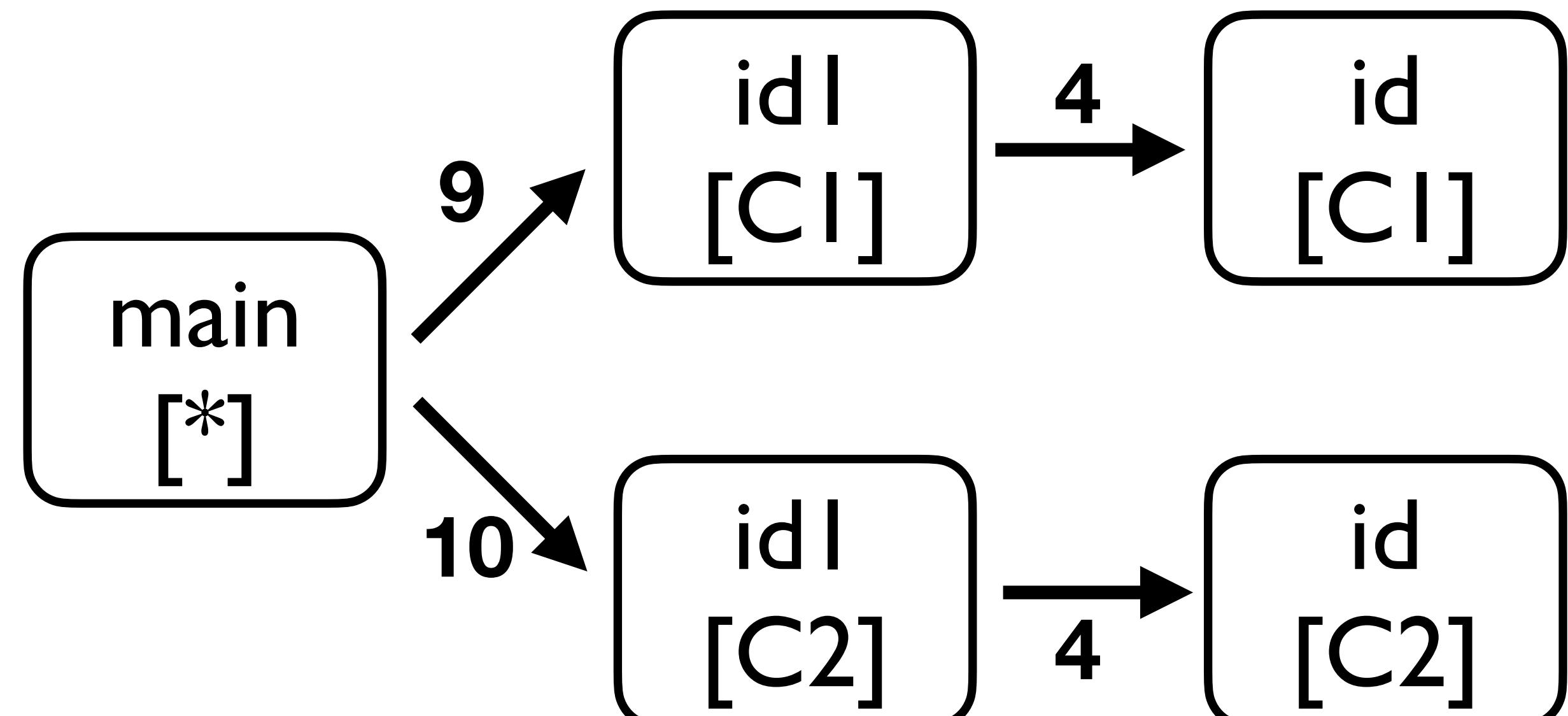


Call-graph of I-CFA

Call-site Sensitivity vs Object Sensitivity

- An example shows the **limitation** of CFA and **strength** of object sensitivity

```
0: class C{  
1:   id(v){  
2:     return v;}  
3:   id1(v){  
4:     return this.id(v);}  
5: }  
6: main(){  
7:   c1 = new C(); //C1  
8:   c2 = new C(); //C2  
9:   a = (A) c1.id1(new A()); //query1  
10:  b = (B) c2.id1(new B()); //query2  
11: }
```

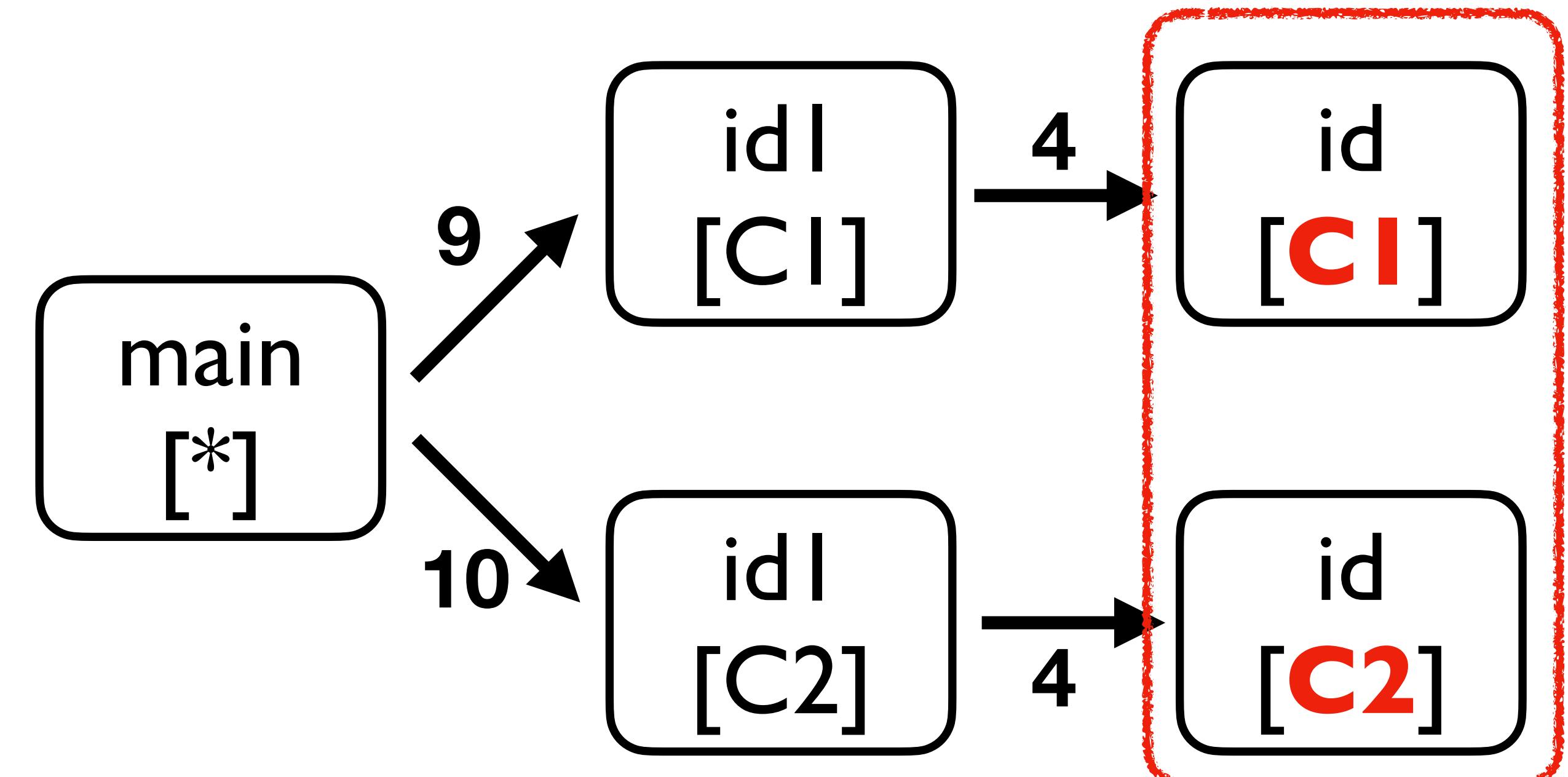


Call-graph of I-Obj

Call-site Sensitivity vs Object Sensitivity

- An example shows the limitation of CFA and strength of object sensitivity

```
0: class C{  
1:     id(v){  
2:         return v;  
3:     }  
4:     id1(v){  
5:         return this.id(v);  
6:     }  
7: }  
main(){  
7:     c1 = new C(); //C1  
8:     c2 = new C(); //C2  
9:     a = (A) c1.id1(new A()); //query1  
10:    b = (B) c2.id1(new B()); //query2  
11: }
```

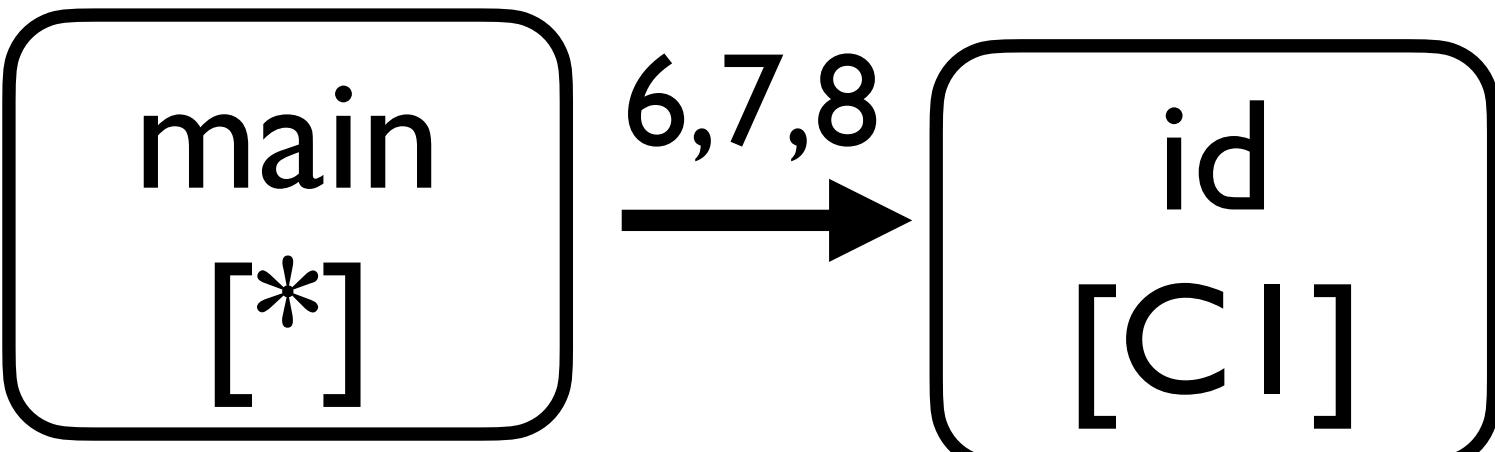


Call-graph of I-Obj

Call-site Sensitivity vs Object Sensitivity

- An example shows the **limitation** of object sensitivity and **strength** of CFA

```
0: class C{  
1:   id(v){  
2:     return v;}  
3: }  
4: main(){  
5:   cl = new C(); //CI  
6:   a = (A) cl.id(new A()); //query1  
7:   b = (B) cl.id(new B()); //query2  
8:   c = (B) cl.id(new C()); //query3  
9: }
```

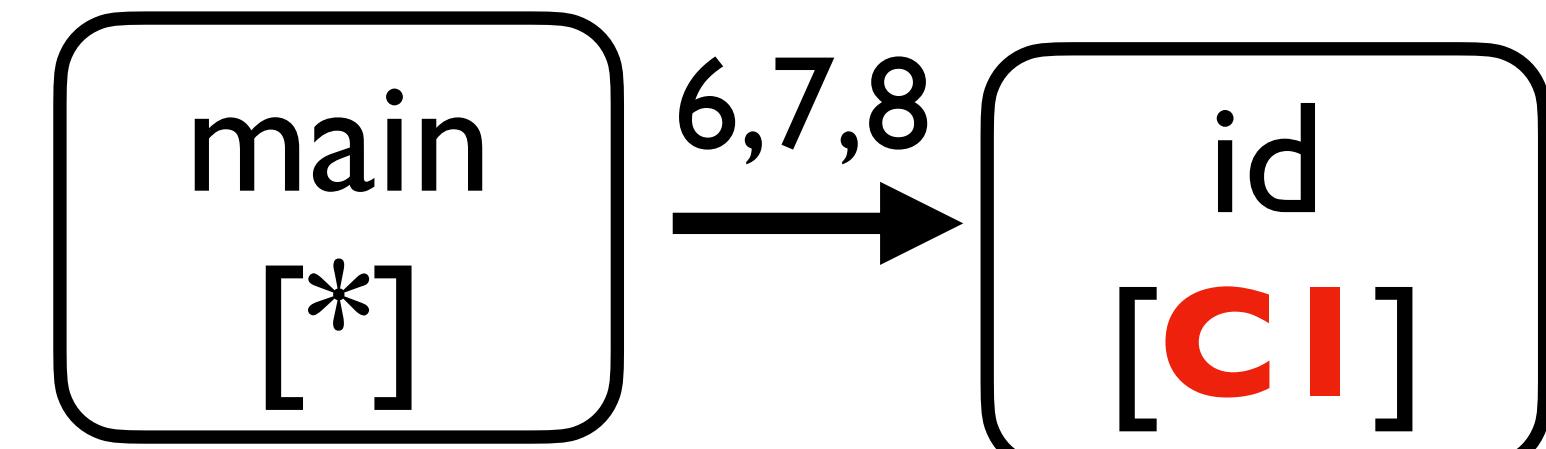


Call-graph of I-Obj

Call-site Sensitivity vs Object Sensitivity

- An example shows the limitation of object sensitivity and strength of CFA

```
0: class C{  
1:   id(v){  
2:     return v;}  
3: }  
4: main(){  
5:   cl = new C();//CI  
6:   a = (A) cl.id(new A());//query1  
7:   b = (B) cl.id(new B());//query2  
8:   c = (B) cl.id(new C());//query3  
9: }
```



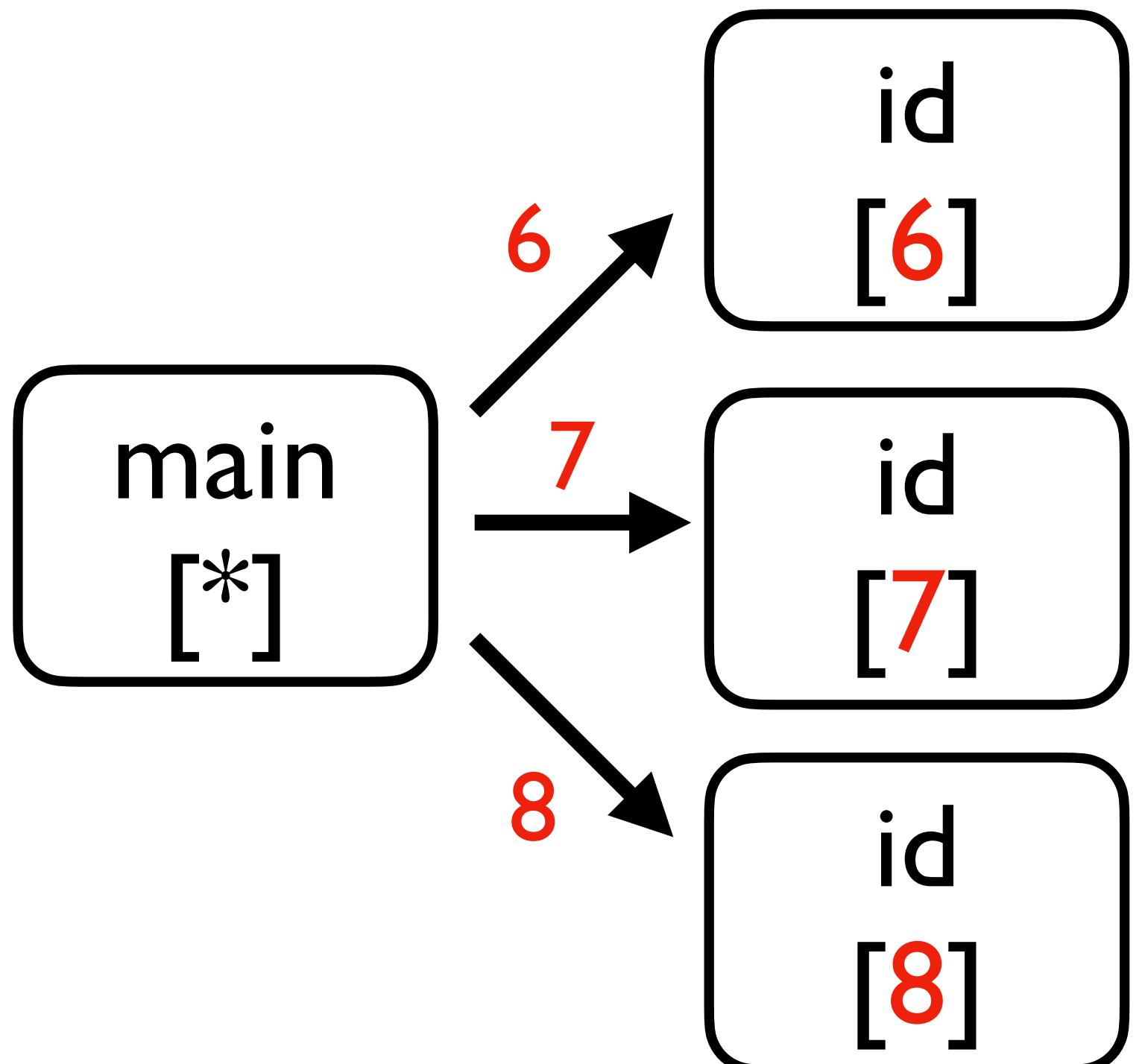
Call-graph of I-Obj

The three method calls share the same receiver object CI

Call-site Sensitivity vs Object Sensitivity

- An example shows the limitation of object sensitivity and strength of CFA

```
0: class C{  
1:   id(v){  
2:     return v;}  
3: }  
4: main(){  
5:   cl = new C(); //C1  
6:   a = (A) cl.id(new A()); //query1  
7:   b = (B) cl.id(new B()); //query2  
8:   c = (B) cl.id(new C()); //query3  
9: }
```



Call-graph of I-CFA

Call-site sensitivity easily separates the three method calls

Call-site Sensitivity vs Object Sensitivity

- Call-site Sensitivity and Object Sensitivity had been **actively compared**

Call-site Sensitivity vs Object Sensitivity

Parameterized Object Sensitivity for Points-to Analysis for Java
ANA MILANOVA
Rensselaer Polytechnic Institute
ATANAS ROUNTEV
Ohio State University
and
BARBARA G. RYDER
Rutgers University

Context-sensitive points-to analysis: is it worth it?*
Ondřej Lhoták^{1,2} and Laurie Hendren²
olhotak@uwaterloo.ca hendren@sable.mcgill.ca
¹ School of Computer Science, University of Waterloo, Waterloo, ON, Canada
² School of Computer Science, McGill University, Montreal, QC, Canada

Evaluating the Benefits of Context-Sensitive Points-to Analysis Using a BDD-Based Implementation
ONDŘEJ LHOTAK
University of Waterloo
and
LAURIE HENDREN
McGill University

Strictly Declarative Specification of Sophisticated Points-to Analyses
Martin Bravenboer Yannis Smaragdakis
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
martin.bravenboer@umass.edu yannis.cs@umass.edu

Abstract. We present the results of an empirical study evaluating the precision of subset-based points-to analysis with respect to the cost of analysis of Java programs of significant size. We compare the use of call site strings as the context abstraction, object sensitivity, and the BDD-based context-sensitive algorithm proposed by Zhu and Calman, and by Whaley and Lam. Our study includes analyses that context-sensitively specialize only pointer variables, as well as ones that also specialize the heap abstraction. We measure both characteristics of the points-to sets themselves, as well as effects on the precision of client analyses. To guide development of efficient analysis implementations, we measure the number of calls to the memory manager, the number of times the number of distinct points-to sets that arise with each context sensitivity variation. To evaluate precision, we measure the size of the call graph in terms of methods and edges, the number of deinitializable call sites, and the number of casts statically provable to be safe. The results of our study indicate that object-sensitive analysis implementations are likely to scale more rapidly and more predictably than the other approaches; that object-sensitive analyses are more precise than comparable variations of the other approaches; that specializing the heap abstraction improves precision more than extending the length of context strings; and that the profusion of cycles in Java call graphs severely reduces precision of analyses that forsake context sensitivity in cyclic regions.

Side-effect analysis. Define *points-to analysis* as pairs of statements that set the value of a memory location at subsequent use that value. The information computed by such analyses has a wide variety of uses in compilers and software tools. This work proposes new versions of these analyses that are based on object-sensitive points-to analysis.

On a set of 23 Java programs, our experiments show that these analyses have comparable cost to a context-insensitive points-to analysis for Java which is based on Andersen's analysis of C. Our results also show that object sensitivity significantly improves the precision of side-effect analysis and can scale efficiently. Specifically, compared to (1) context-insensitive analysis, and (2) context-sensitive points-to analysis that models control using the invoking call site, these experiments demonstrate that object-sensitive analyses can achieve substantial precision improvement, while at the same time remaining efficient and practical.

A preliminary version of this article appeared in *Proceedings of the International Symposium on Software Testing and Analysis* (July), 2002, pp. 1–11. This work was supported in part by NSERC and the Canadian Foundation (NSF) grant OCE-0004988. Author's address: A. Rountev, Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180; email: milanova@rpi.edu; A. Rountev, Department of Computer Science and Engineering, Ohio State University, 2015 Neil Avenue, Columbus, OH 43210; email: rountev@cs.ohio-state.edu; B. G. Ryder, Department of Computer Sciences, Rutgers University, 100 University Avenue, Newark, NJ 07102; email: ryder@cs.rutgers.edu. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036, USA; fax: +1 (212) 889-0481; or permissions@acm.org. © 2008 ACM 1049-312X/08/09-ART3 \$5.00 DOI 10.1145/1391984.1391987 http://doi.acm.org/10.1145/1391984.1391987

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors; D.3.3 [Programming Languages]: Language Constructs and Features
General Terms: Languages, Design, Experimentation, Measurement
Additional Key Words and Phrases: Interprocedural program analysis, context sensitivity, binary decision diagrams, points-to analysis, call graph construction, cast safety analysis

ACM Reference Format:
Lhoták, O. and Hendren, L. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Method.* 18, 1, Article 3 (September 2008), 53 pages. DOI = 10.1145/1391984.1391987 http://doi.acm.org/10.1145/1391984.1391987

This is a revised and extended version of an article which appeared in *Proceedings of the 15th International Conference on Compiler Construction*, Lecture Notes in Computer Science, vol. 3923. Springer, 47–64.
Authors' addresses: O. Lhoták, D. R. Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON, N2L 3G1, Canada; L. Hendren, School of Computer Science, McGill University, 3480 University Street, Room 318, Montreal, QC, H3A 2A7, Canada.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036, USA; fax: +1 (212) 889-0481; or permissions@acm.org. © 2008 ACM 1049-312X/08/09-ART3 \$5.00 DOI 10.1145/1391984.1391987 http://doi.acm.org/10.1145/1391984.1391987

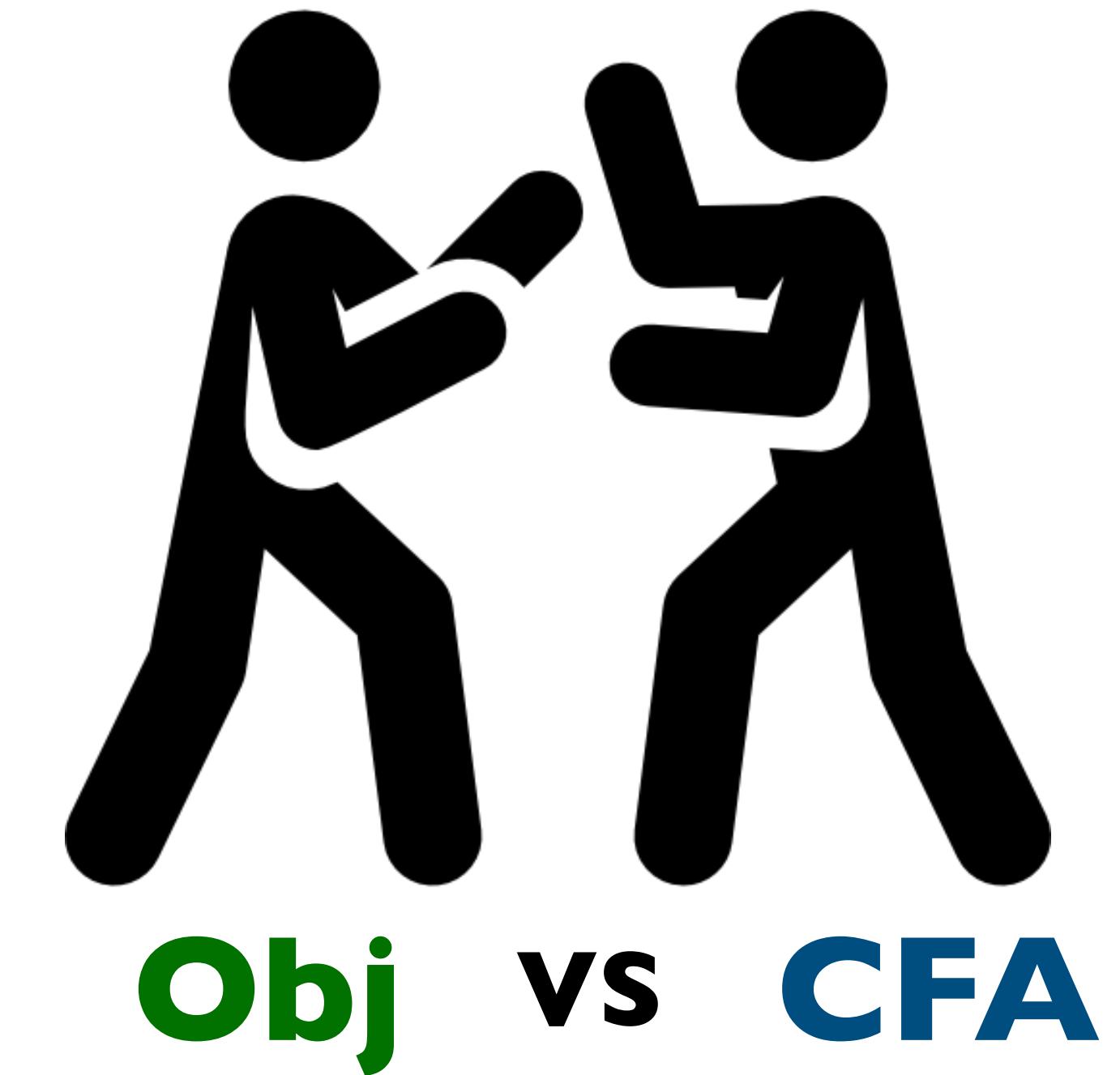
Categories and Subject Descriptors: F.3.2 [Logics and Problem Description]: Logic of Programs—Program Analysis; F.3.3 [Programming Languages]: Programming Languages—Program Analysis; D.1.6 [Programming Techniques]: Logic Programming
General Terms: Algorithms, Languages, Performance

1. Introduction
Points-to (or pointer) analysis intends to answer the question “What objects can a program point to?” This question forms the basis for practically all higher-level program analyses. It is, thus, not surprising that a wealth of research has been devoted to this problem, particularly in the context of static analysis. Context-sensitive analysis is the most challenging class of points-to analyses. Context-sensitive analysis approaches qualify the analysis facts with a context abstraction, which captures a static notion of the dynamic context of a method. Typical contexts include abstractions of method calls (e.g., a *call-site sensitive* analysis—the traditional meaning of “context-sensitive”) or receiver objects (or an *object-sensitive* analysis).

In this work we present *Door*, a general and versatile points-to analysis framework, designed to be the most precise context-sensitive analysis reported in the literature. *Door* implements a range of algorithms, including context-insensitive, call-site sensitive, and object-sensitive analyses, all specified modularly as variations on a common code base. Compared to the prior state of the art, *Door* often achieves speedups of orders-of-magnitude for several important analyses. The main elements of our approach are the use of the Data-Flow algorithm for specifying context abstractions, the use of a generic optimizer for the *Datalog* program. The use of *Datalog* for program analysis (both low-level [13,23,29] and high-level [6,9]) is far from new. Our novel optimization approach, however, accounts for several orders of magnitude of performance improvement. Our optimizer handles over 1000 times more points-to analysis facts than the state-of-the-art alternative, *Door*’s optimizer, and our optimizations fit well the approach of handling program facts as a database, by specifically targeting the indexing scheme and the incremental evaluation of *Datalog* implementations. Furthermore, we extend *Door*’s analysis, enabling declaratively the logic required for call graph construction as well as for handling the full semantic complexity of the Java language (e.g., static initialization, finalization, references, objects, threads, exceptions, reflection, etc.). This makes our analysis amenable to reuse, as well as being both efficient and easy to tune. Generally, our work is a strong data point in support of declarative languages: we argue that prohibitively much human effort is required for implementing and optimizing complex mutually-recursive definitions at an operational level of abstraction. On the other

* This work was supported, in part, by NSERC and an IBM Ph.D. Fellowship.

ACM Transactions on Software Engineering and Methodology, Vol. 14, No. 1, January 2005, Pages 1–41.



1981

2002

2010

2022

Call-site Sensitivity vs Object Sensitivity

- Object Sensitivity **outperformed** call-site sensitivity

Call-site Sensitivity vs Object Sensitivity

Parameterized Object Sensitivity for Points-to Analysis for Java

ANA MILANOVA
Rensselaer Polytechnic Institute
ATANAS ROUNTEV
Ohio State University
and
BARBARA G. RYDER
Rutgers University

The goal of points-to analysis for Java is to determine the set of objects pointed to by a reference variable or a reference object field. We present object sensitivity, a new form of context sensitivity for flow-insensitive points-to analysis for Java. The key idea of our approach is to analyze a method separately for each of the object names that represent run-time objects on which this method may be invoked. To ensure flexibility and practicality, we propose a parameterization framework that allows analysis designers to control the tradeoffs between cost and precision in the object-sensitive analysis.

Side-effect analysis. Define-and-set identifies pairs of statements that set the value of a memory location and subsequently use that value. The information computed by such analyses has a wide variety of uses in compilers and software tools. This work proposes new versions of these analyses that are based on object-sensitive points-to analysis.

A preliminary version of this article appeared in Proceedings of the International Symposium on Software Testing and Analysis (July), 2002, pp. 1–11. This work was supported in part by National Science Foundation (NSF) grant CCR-0009888. Author's address: A. Rountev, Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180; email: milanova@rpi.edu; A. Rountev, Department of Computer Science and Engineering, Ohio State University, 2015 Neil Avenue, Columbus, OH 43210; email: rountev@cs.ohio-state.edu; B. G. Ryder, Department of Computer Sciences, Rutgers University, 100 University Avenue, Piscataway, NJ 08854; email: ryder@cs.rutgers.edu. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be granted separately.

Context-sensitive points-to analysis: is it worth it?*

Ondřej Lhoták^{1,2} and Laurie Hendren²
olhotak@uwaterloo.ca hendren@sable.mcgill.ca

¹ School of Computer Science, University of Waterloo, Waterloo, ON, Canada
² School of Computer Science, McGill University, Montreal, QC, Canada

We present the results of an empirical study evaluating the precision of subset-based points-to analysis with respect to the cost of analysis for Java programs of significant size. We compare the use of call site strings as the context abstraction, object sensitivity, and the BDD-based context-sensitive algorithm proposed by Zhu and Calman, and by Whaley and Lam. Our study includes analyses that context-sensitively specialize only pointer variables, as well as ones that also specialize the heap abstraction. We measure both characteristics of the points-to sets themselves, as well as effects on the precision of client analyses. To guide development of efficient analysis implementations, we measure the number of casts in the target program, and the number of times that the number of distinct points-to sets that arise with each context sensitivity variation. To evaluate precision, we measure the size of the call graph in terms of methods and edges, the number of deinitializable call sites, and the number of casts statically proved to be safe.

The results of our study indicate that object-sensitive analysis implementations are likely to scale better and more predictably than the other approaches; that object-sensitive analyses are more precise than comparable variations of the other approaches; that specializing the heap abstraction improves precision more than extending the length of context strings; and that the profusion of cycles in Java call graphs severely reduces precision of analyses that forsake context sensitivity in cyclic regions.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms: Languages, Design, Experimentation, Measurement

Additional Key Words and Phrases: Interprocedural program analysis, context sensitivity, binary decision diagrams, points-to analysis, call graph construction, cast safety analysis

ACM Reference Format:

Lhoták, O. and Hendren, L. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Method. 18, 1, Article 3 (September 2008), 53 pages. DOI = 10.1145/1391984.1391987. http://doi.acm.org/10.1145/1391984.1391987

Evaluating the Benefits of Context-Sensitive Points-to Analysis Using a BDD-Based Implementation

ONDŘEJ LHOTÁK
University of Waterloo
and
LAURIE HENDREN
McGill University

We present Poosla, a framework of BDD-based context-sensitive points-to and call graph analyses for Java, as well as client analyses that use their results. Poosla supports several variations of context-sensitive analyses, including call site strings and object sensitivity, and context-sensitively specializes both pointer variables and the heap abstraction. We empirically evaluate the precision of these context-sensitive analyses on significant Java programs. We find that object-sensitive analyses are more precise than comparable variations of the other approaches, and that specializing the heap abstraction improves precision more than extending the length of context strings.

Categories and Subject Descriptors: F.3.3 [Logic and Programming]: Programming Languages—Program Analysis; F.3.2 [Logics and Proof]: Logic Programming

General Terms: Languages, Design, Experimentation, Measurement

Additional Key Words and Phrases: Interprocedural program analysis, context sensitivity, binary decision diagrams, points-to analysis, call graph construction, cast safety analysis

ACM Reference Format:

Lhoták, O. and Hendren, L. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. ACM Trans. Softw. Eng. Method. 18, 1, Article 3 (September 2008), 53 pages. DOI = 10.1145/1391984.1391987. http://doi.acm.org/10.1145/1391984.1391987

Strictly Declarative Specification of Sophisticated Points-to Analyses

Martin Bravenboer¹ Yannis Smaragdakis²
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
martin.bravenboer@umass.edu yannis.cs@umass.edu

We present Door, a framework for points-to analysis of Java programs. Door builds on the idea of specifying pointer analysis algorithms declaratively, using DataLog: a logic-based language for specifying pointer analysis. We show how the declarative approach rather than past work by defining the full end-to-end analysis in DataLog and optimizing aggressively using a novel technique specifically targeting highly recursive pointer analysis.

As a result, Door achieves several benefits, including full order-of-magnitude improvements in runtime. We compare Door with Lhoták and Hendren's Poosla, which defines the state-of-the-art for context-sensitive analyses. For the example of points-to analysis, Door is 10x faster than Poosla for a 1-call-site sensitive analysis of the DaCapo benchmarks, with lower but still substantial speedups for other important analyses. Additionally, Door achieves precision guarantees that are comparable with Lhoták and Whaley's Vals, yet directly addressing open problems in past literature. Finally, our implementation is modular and can be easily configured to analyses with a wide range of characteristics, largely to its declarativeness.

In this work we present Door, a general and versatile pointer analysis framework that builds on the most precise context-sensitive analyses reported in the literature. The state-of-the-art for context-sensitive analyses. For the example of points-to analysis, Door is 10x faster than Poosla for a 1-call-site sensitive analysis of the DaCapo benchmarks, with lower but still substantial speedups for other important analyses. Additionally, Door achieves precision guarantees that are comparable with Lhoták and Whaley's Vals, yet directly addressing open problems in past literature. Finally, our implementation is modular and can be easily configured to analyses with a wide range of characteristics, largely to its declarativeness.

Categories and Subject Descriptors: F.3.2 [Logics and Proof]: Logic Programming—Program Analysis; F.3.3 [Logic and Programming Languages]: Programming Languages—Program Analysis; D.1.6 [Programming Techniques]: Logic Programming

General Terms: Algorithms, Languages, Performance

1. Introduction

Points-to (or pointer) analysis intends to answer the question “What objects can a program variable point to?” This question forms the basis for practically all higher-level program

...

Obj

CFA

1981

2002

2010

2022

16

Call-site Sensitivity vs Object Sensitivity

- Lectures have taught the **superiority** of object sensitivity

Object-Sensitivity

- The dominant flavor of context-sensitivity for object-oriented languages.
- It uses object abstractions (i.e. allocation sites) as well as qualifying a method's local variables with the allocation site of the receiver object of the method call.

```
program
class S {
    Object id(Object a) { return a; }
    Object id2(Object a) { return id(a); }
}
class C extends S {
    void fun1() {
        Object a1 = new A1();
        Object b1 = id2(a1);
    }
}
class D extends S {
    void fun2() {
        Object a2 = new A2();
        Object b2 = id2(a2);
    }
}
```

The context of `m` is the allocation site of `b`.

Hakjoo Oh AAA616 2019 Fall, Lecture 8

**Object-Sensitivity
(vs. call-site sensitivity)**

```
program
class S {
    Object id(Object a) { return a; }
    Object id2(Object a) { return id(a); }
}
class C extends S {
    void fun1() {
        Object a1 = new A1();
        Object b1 = id2(a1);
    }
}
class D extends S {
    void fun2() {
        Object a2 = new A2();
        Object b2 = id2(a2);
    }
}
```

Yannis Smaragdakis
University of Athens

Object-sensitive pointer analysis

- Milanova, Rountev, and Ryder. *Parameterizing sensitivity for points-to analysis for Java*. ACM SIGART Eng. Methodol., 2005.
 - Context-sensitive interprocedural pointer analysis
 - For context, use stack of receiver objects
 - (More next week?)
- Lhotak and Hendren. *Context-sensitive pointer analysis: Is it worth it?* CC 06
 - Object-sensitive pointer analysis more precise than call-site sensitivity
 - Likely to scale better

Lecture Notes:
Pointer Analysis

15-819O: Program Analysis
Jonathan Aldrich
jonathan.aldrich@cs.cmu.edu

Lecture 9

1 Motivation for Pointer Analysis

In programs with pointers, program analysis can become more complex. Consider constant-propagation analysis of the following program:

```
1: z := 1
2: p := &z
3: *p := 2
4: print z
```

In order to analyze this program correctly we must be aware that instruction 3 `p` points to `z`. If this information is available we can analyze the function as follows:

```
fCP[*p := y](σ) = [z ↦ σ(y)]σ where must-point-to(p)
```

When we know exactly what a variable `z` points to, we say that `z` has a **must-point-to** information, and we can perform a **strong update** of variable `z`, because we know with confidence that assigning to `z` will update the variable `z` to point to `y`. A technicality in the rule is quantifying over all `z` such that `*p` points to `z`. How is this possible? It is not possible in C or Java, which are languages with pass-by-reference, for example C++, it is possible because names for the same location are in scope.

Of course, it is also possible that we are uncertain to which of the two distinct locations `p` points. For example:

now
the essence of knowledge

Pointer Analysis

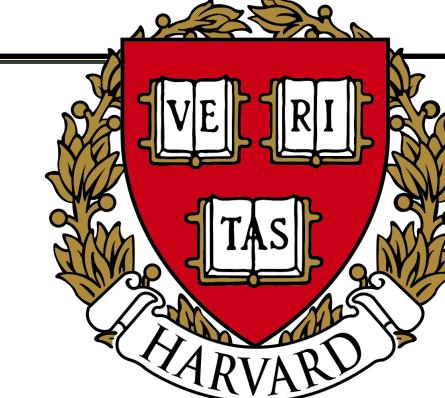
Yannis Smaragdakis
University of Athens
smaragd@di.uoa.gr

George Balatsouras
University of Athens
gbalats@di.uoa.gr

• • •



National and Kapodistrian
University of Athens



Carnegie
Mellon
University



Obj

1981

2002

2010

2022

Call-site Sensitivity vs Object Sensitivity

- Lectures have taught the **superiority** of object sensitivity



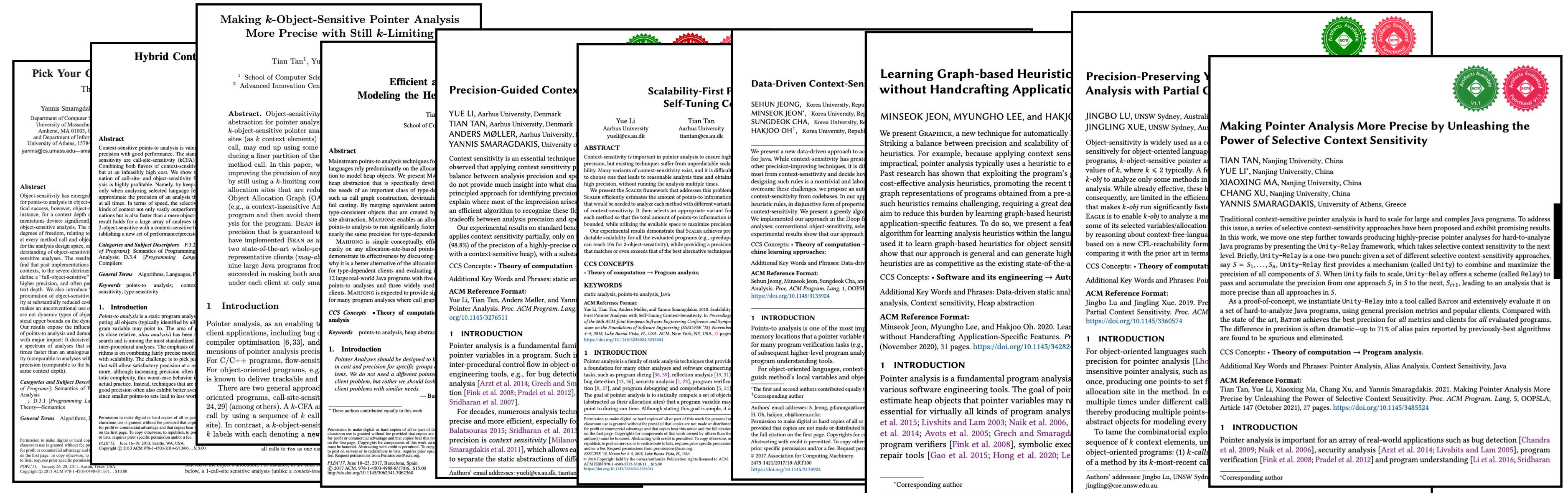
Obj



Call-site Sensitivity vs Object Sensitivity

- Researches focused on improving Object Sensitivity

Researches on Object Sensitivity



1981

2002

2010

Call-site Sensitivity vs Object Sensitivity

- Call-site Sensitivity has been ignored

“... call-site-sensitivity is less important than others ...”
- Jeon et al. [2019]



1981

2002

2010

Call-site Sensitivity vs Object Sensitivity

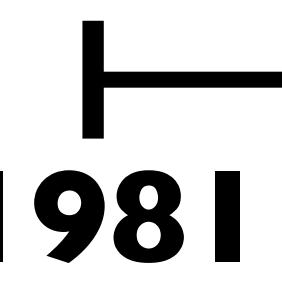
- Call-site Sensitivity has been ignored

“... call-site-sensitivity is less important than others ...”
- [Jeon] et al. [2019]

I also strongly dismissed call-site sensitivity



CFA

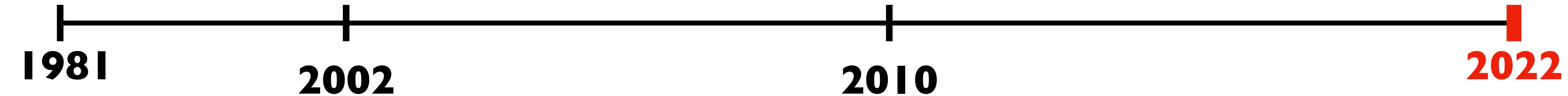


2002

2010

Call-site Sensitivity vs Object Sensitivity

Currently, call-site sensitivity is known as a bad context



Call-site Sensitivity vs Object Sensitivity

A technique **context tunneling** is proposed



Context tunneling can improve both
call-site sensitivity and object sensitivity

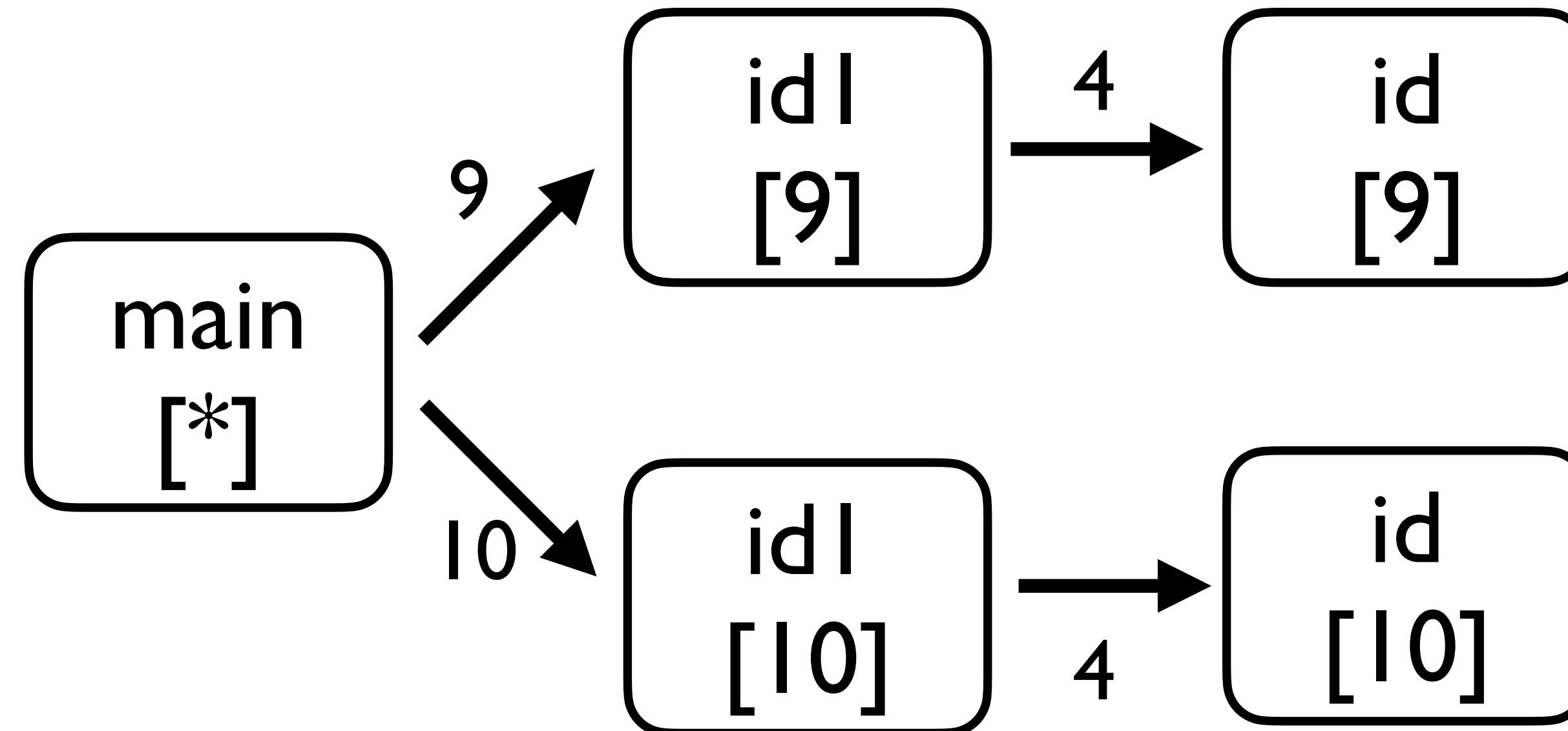
Jeon et al. [2018]



Call-site Sensitivity vs Object Sensitivity

- **Context tunneling** can remove the limitation of call-site sensitivity

```
0: class C{  
1:   id(v){  
2:     return v;  
3:   id1(v){  
4:     return id0(v);}  
5: }  
6: main(){  
7:   c1 = new C();//C1  
8:   c2 = new C();//C2  
9:   a = (A) c1.id1(new A());//query1  
10:  b = (B) c2.id1(new B());//query2  
11: }
```

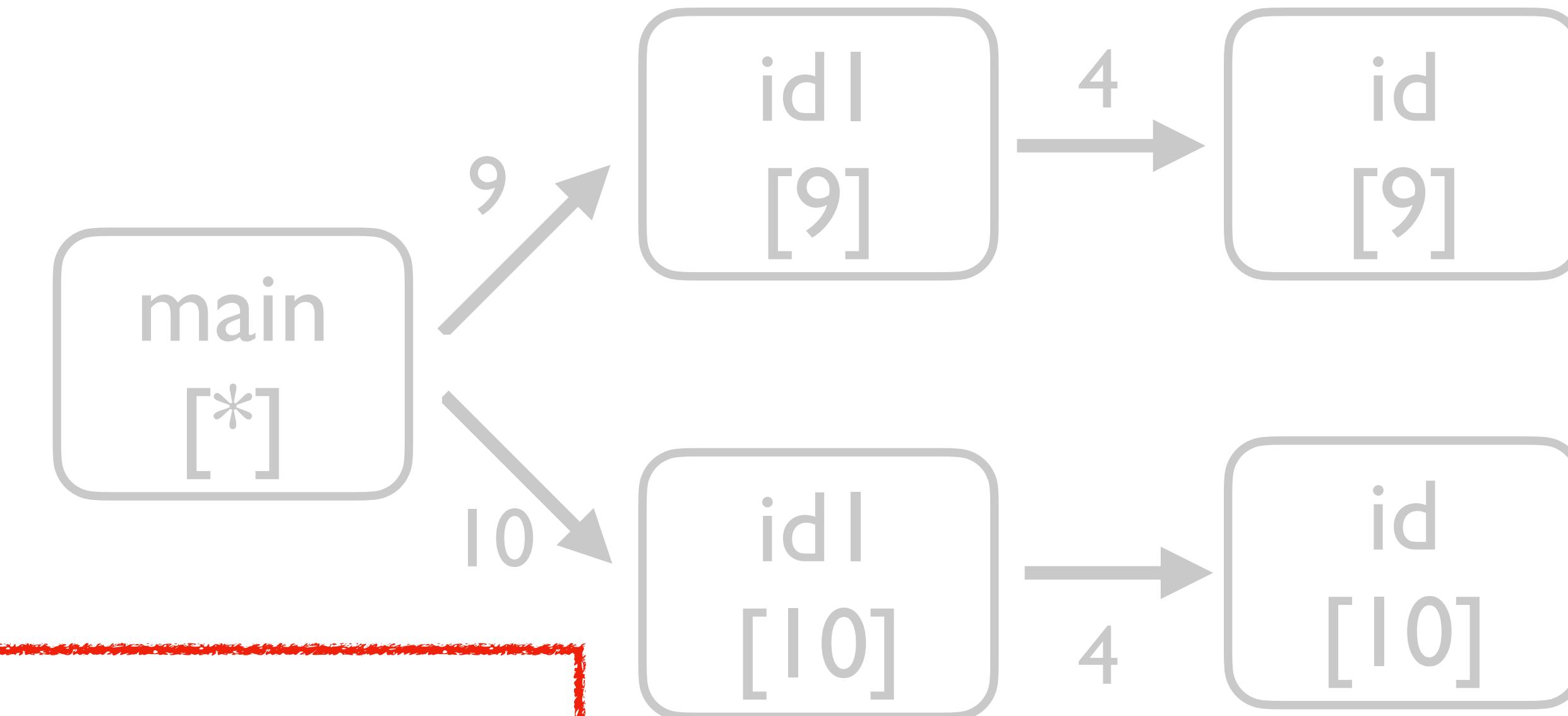


I-CFA with context tunneling
(T= {4})

Call-site Sensitivity vs Object Sensitivity

- Context tunneling can remove the limitation of call-site sensitivity

```
0: class C{  
1:     id(v){  
2:         return v;  
3:     }  
4:     idI(v){  
5:         return id0(v);}  
6: }  
7: main(){  
8:     cl = new C();  
9: }
```



Tunneling abstraction:

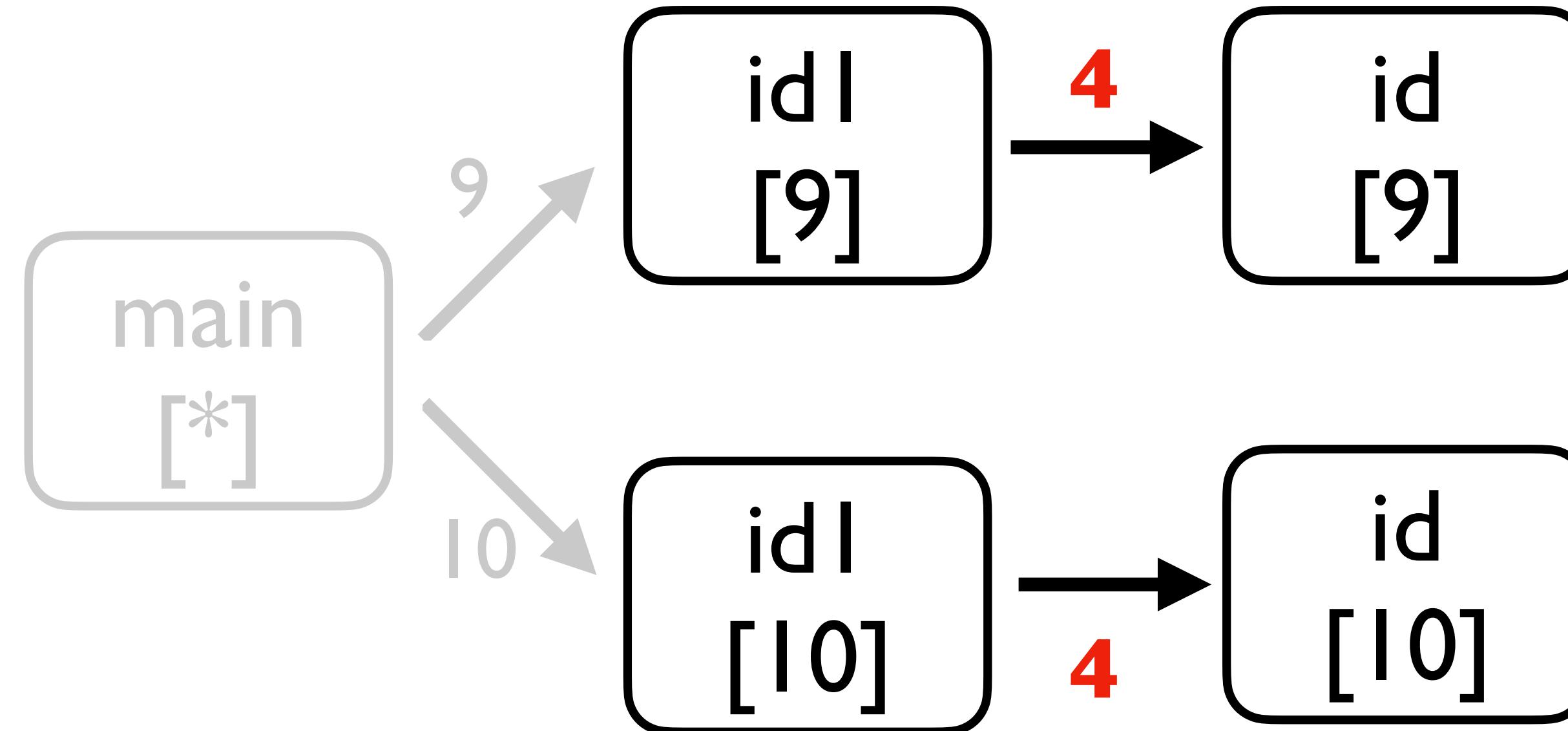
Determines where to apply context tunneling

with context tunneling
 $(T = \{4\})$

Call-site Sensitivity vs Object Sensitivity

- Context tunneling can remove the limitation of call-site sensitivity

```
0: class C{  
1:   id(v){  
2:     return v;  
3:   }  
4:   id1(v){  
5:     return id0(v);}  
6: }  
6: main(){  
7:   c1 = new C(); //C1  
8:   c2 = new C(); //C2  
9:   a = (A) c1.id1(new A()); //query1  
10:  b = (B) c2.id1(new B()); //query2  
11: }
```



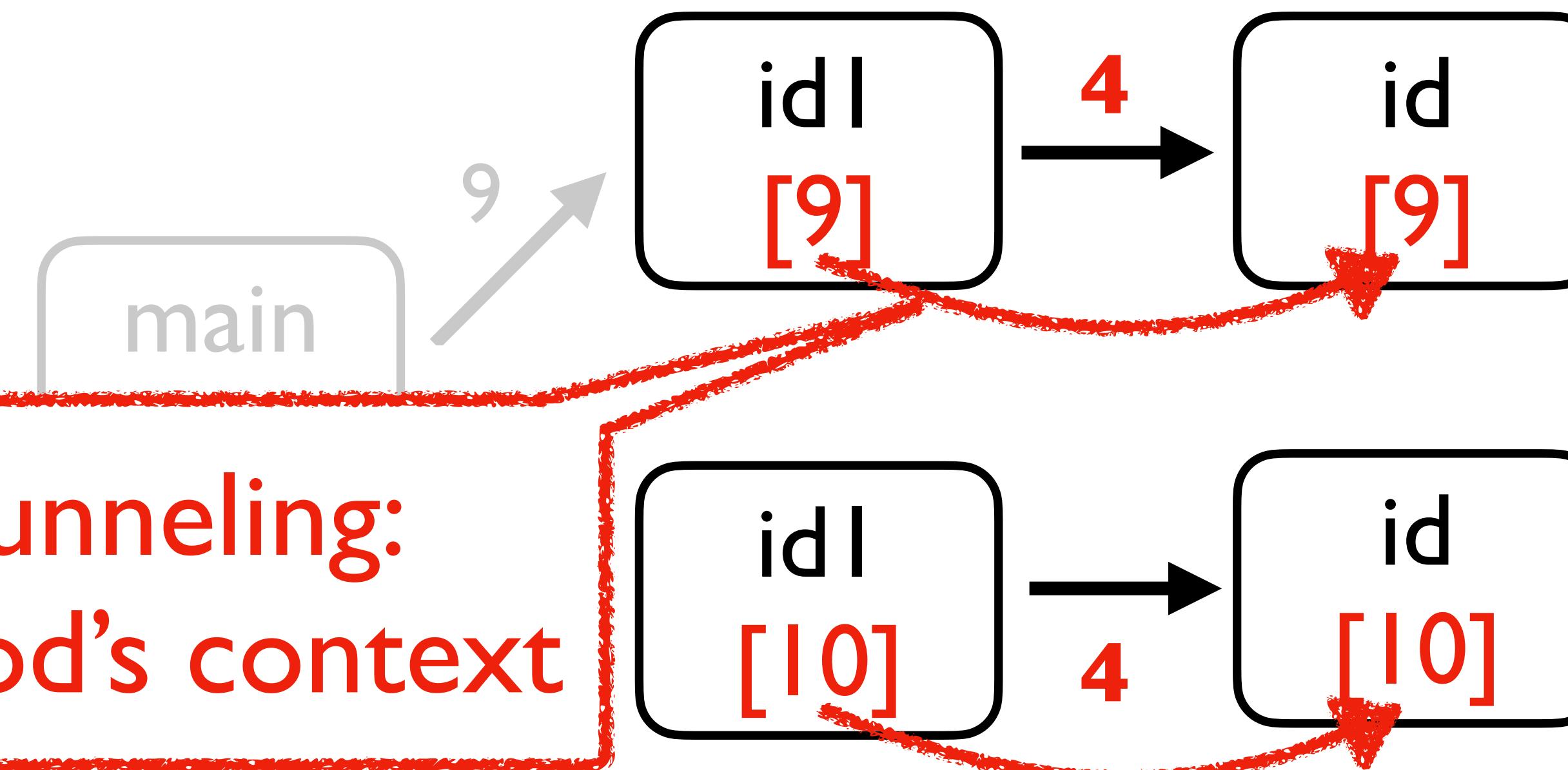
I-CFA with context tunneling
(T= {4})

Unimportant call-sites that should not be used as context elements

Call-site Sensitivity vs Object Sensitivity

- Context tunneling can remove the limitation of call-site sensitivity

```
0: class C{  
1:   id(v){  
2:     return v;  
3:   }  
4:   idI(v){  
5:     return id0(v);  
6:   }  
7:   main(){  
8:     c1 = new C();  
9:     a = (A) c1.idI(new A()); //query1  
10:    b = (B) c2.idI(new B()); //query2  
11:  }
```



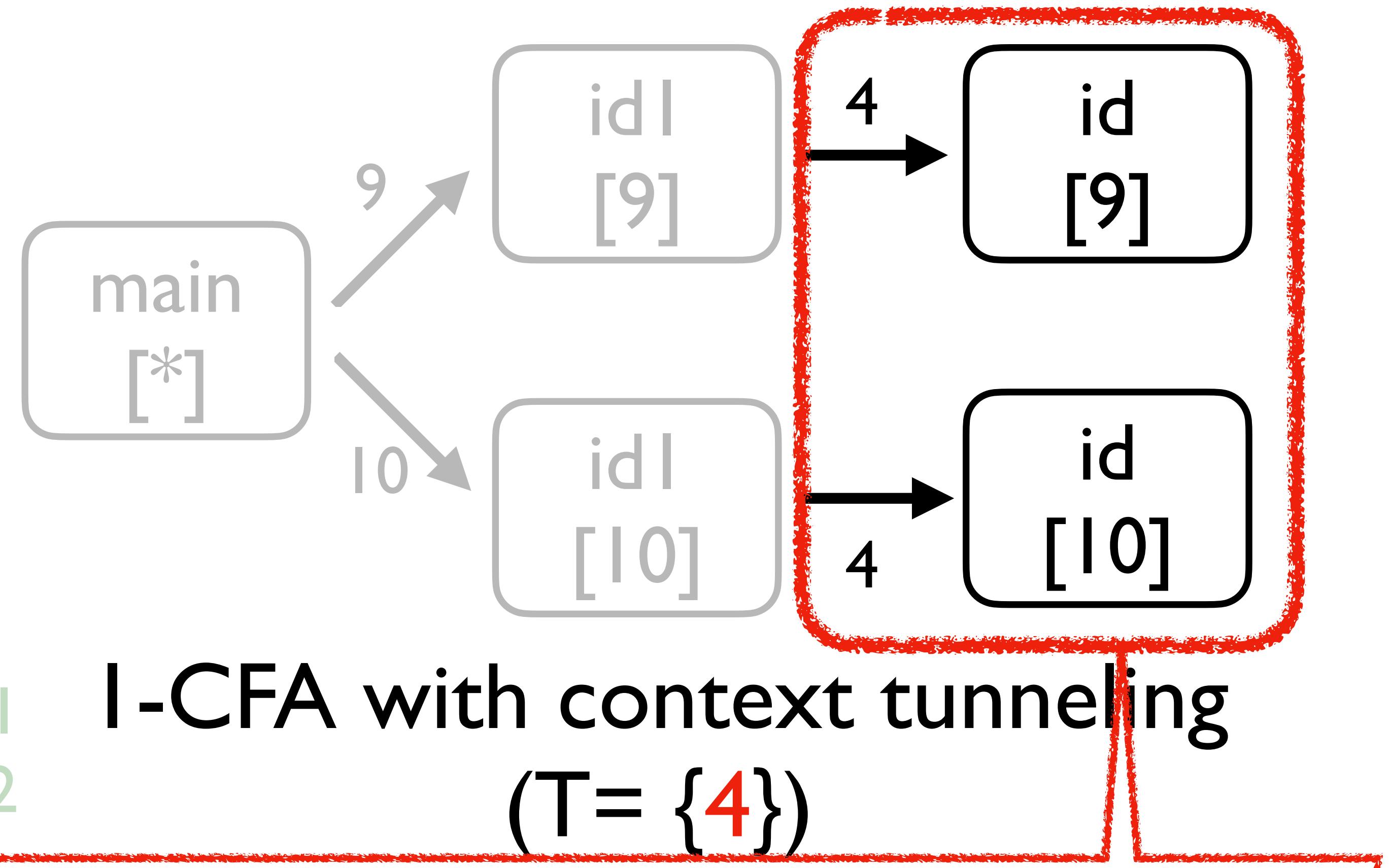
Apply context tunneling:
Inherit caller method's context

I-CFA with context tunneling
(T= {4})

Call-site Sensitivity vs Object Sensitivity

- Context tunneling can remove the limitation of call-site sensitivity

```
0: class C{  
1:   id(v){  
2:     return v;  
3:   id1(v){  
4:     return id0(v);}  
5: }  
6: main(){  
7:   c1 = new C();//C1  
8:   c2 = new C();//C2  
9:   a = (A) c1.id1(new A());//query1  
10:  b = (B) c2.id1(new B());//query2  
11: }
```

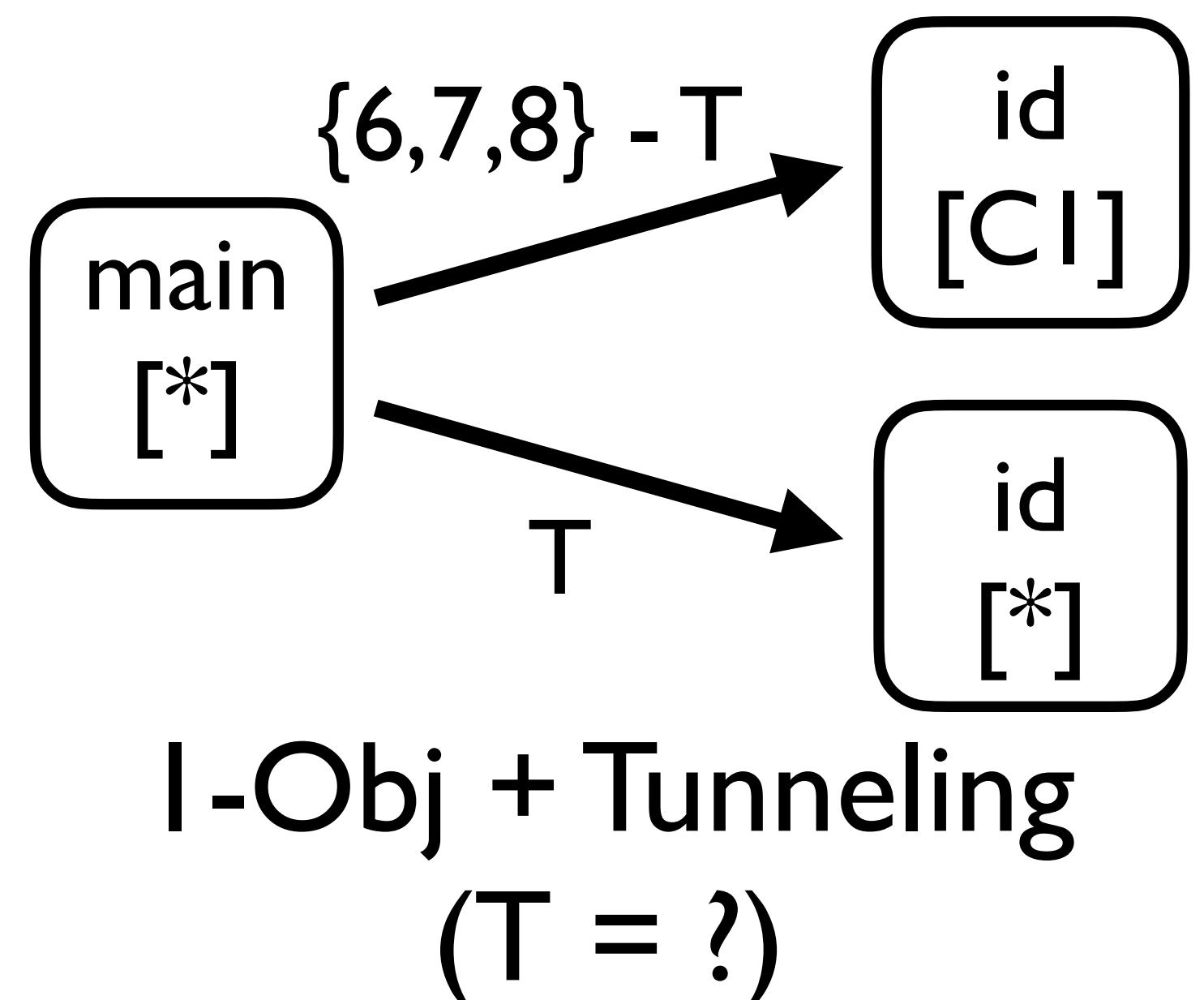


With tunneling, I-CFA separates the nested method calls

Call-site Sensitivity vs Object Sensitivity

- Object sensitivity still suffers from its limitation

```
0: class C{  
1:   id(v){  
2:     return v;}  
3: }  
4: main(){  
5:   cl = new C(); //CI  
6:   a = (A) cl.id(new A());  
7:   b = (B) cl.id(new B());  
8:   c = (B) cl.id(new C());  
9: }
```

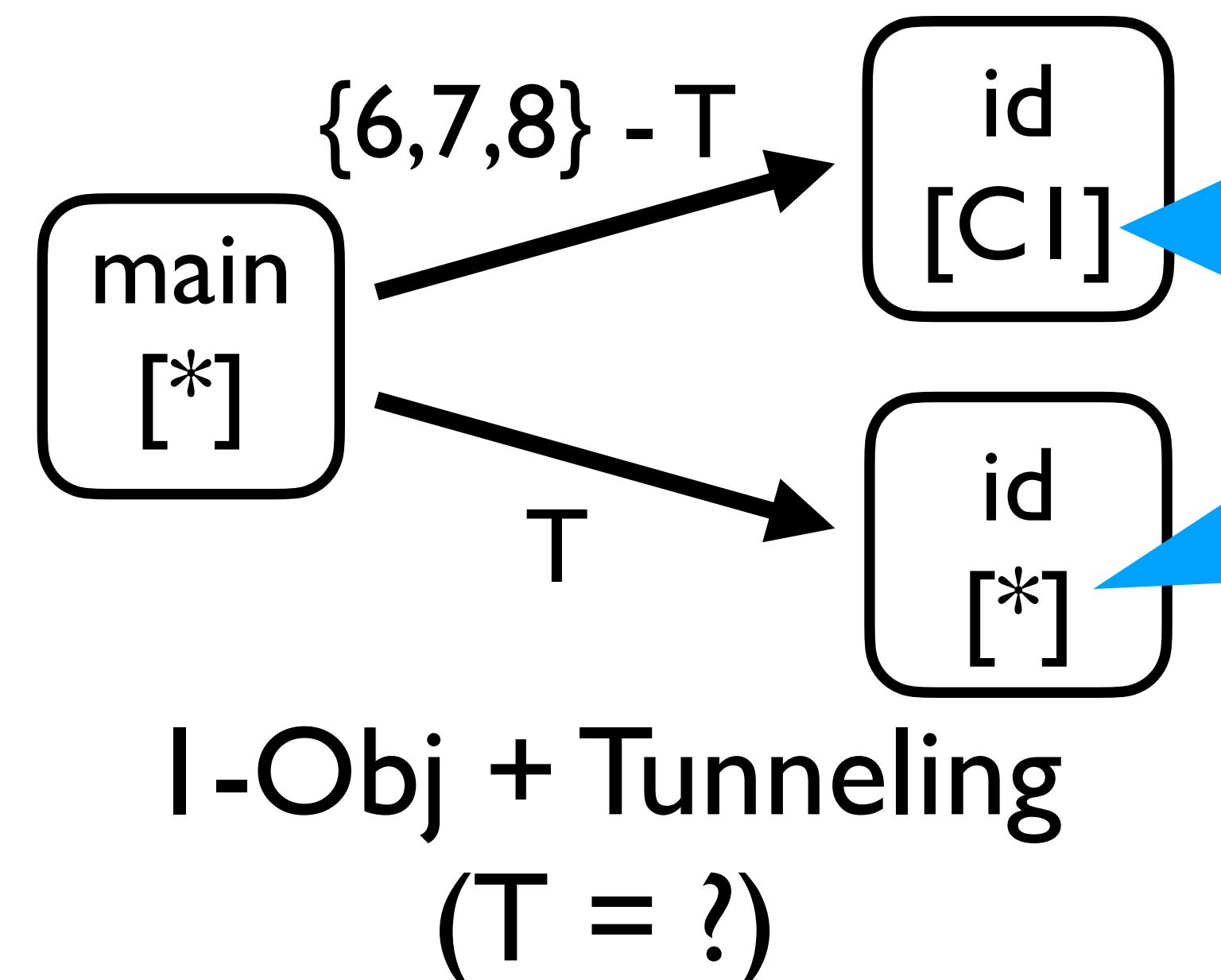


Call-graph of I-Obj with tunneling T

Call-site Sensitivity vs Object Sensitivity

- Object sensitivity still suffers from its limitation

```
0: class C{  
1:   id(v){  
2:     return v;}  
3: }  
4: main(){  
5:   cl = new C(); //CI  
6:   a = (A) cl.id(new A());  
7:   b = (B) cl.id(new B());  
8:   c = (B) cl.id(new C());  
9: }
```

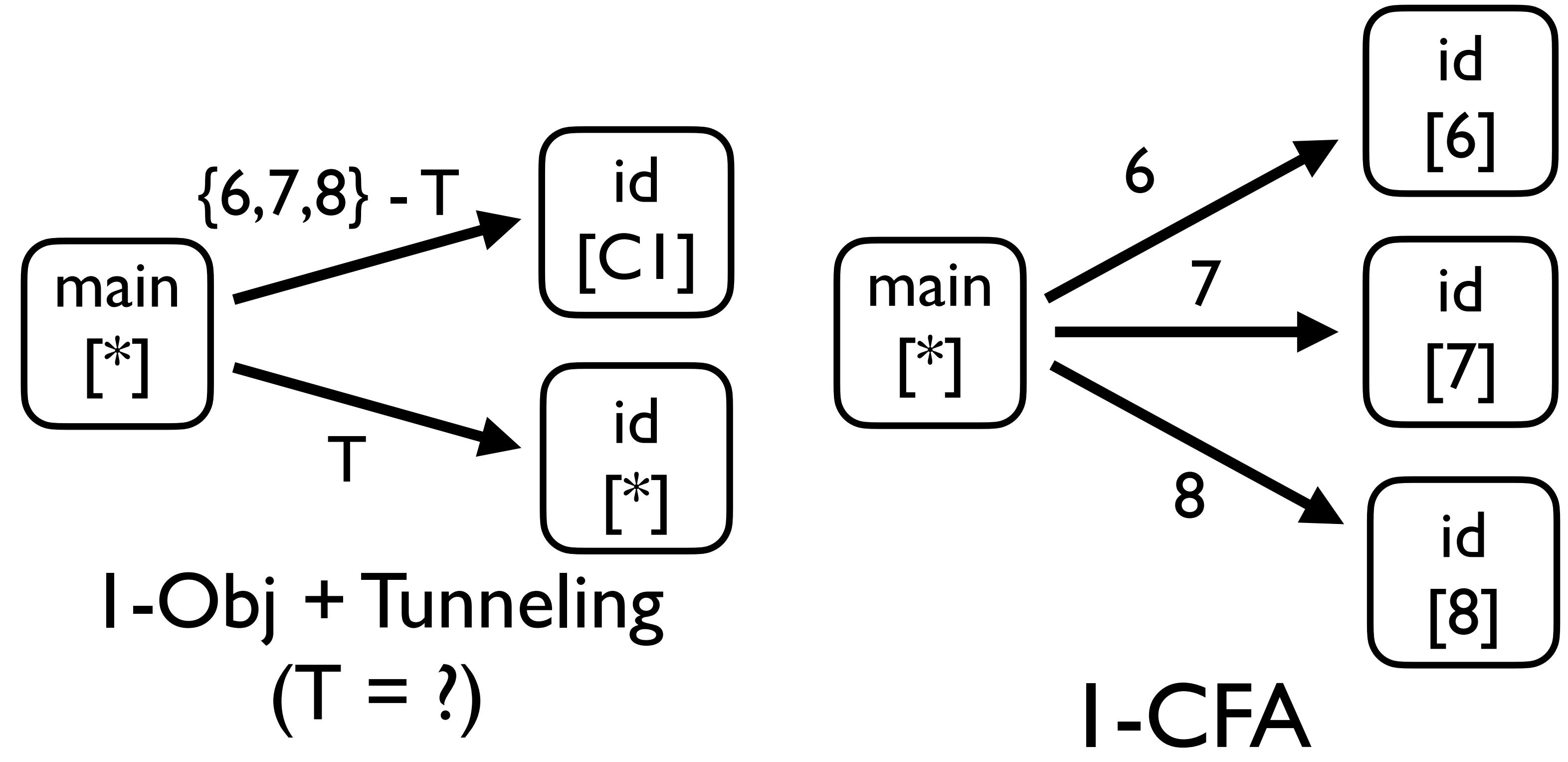


Unable to separate the three method calls with the two contexts

Call-site Sensitivity vs Object Sensitivity

- Object sensitivity still suffers from its limitation

```
0: class C{  
1:   id(v){  
2:     return v;}  
3: }  
4: main(){  
5:   cl = new C(); //CI  
6:   a = (A) cl.id(new A());  
7:   b = (B) cl.id(new B());  
8:   c = (B) cl.id(new C());  
9: }
```



Call-site sensitivity easily separates the three method calls

Call-site Sensitivity vs Object Sensitivity

- Object sensitivity still suffers from its limitation.

Observation

When context tunneling is included

- Limitation of call-site sensitivity is **removed**
- Limitation of object sensitivity is **not removed**

```
0: c
1:
2: }
3:
4: n
5:
6: a
7: b = (B) cl.id(new B());
8: c = (B) cl.id(new C());
9: }
```

Call-site Sensitivity vs Object Sensitivity

- Object sensitivity still suffers from its limitation.

Observation

When context tunneling is included

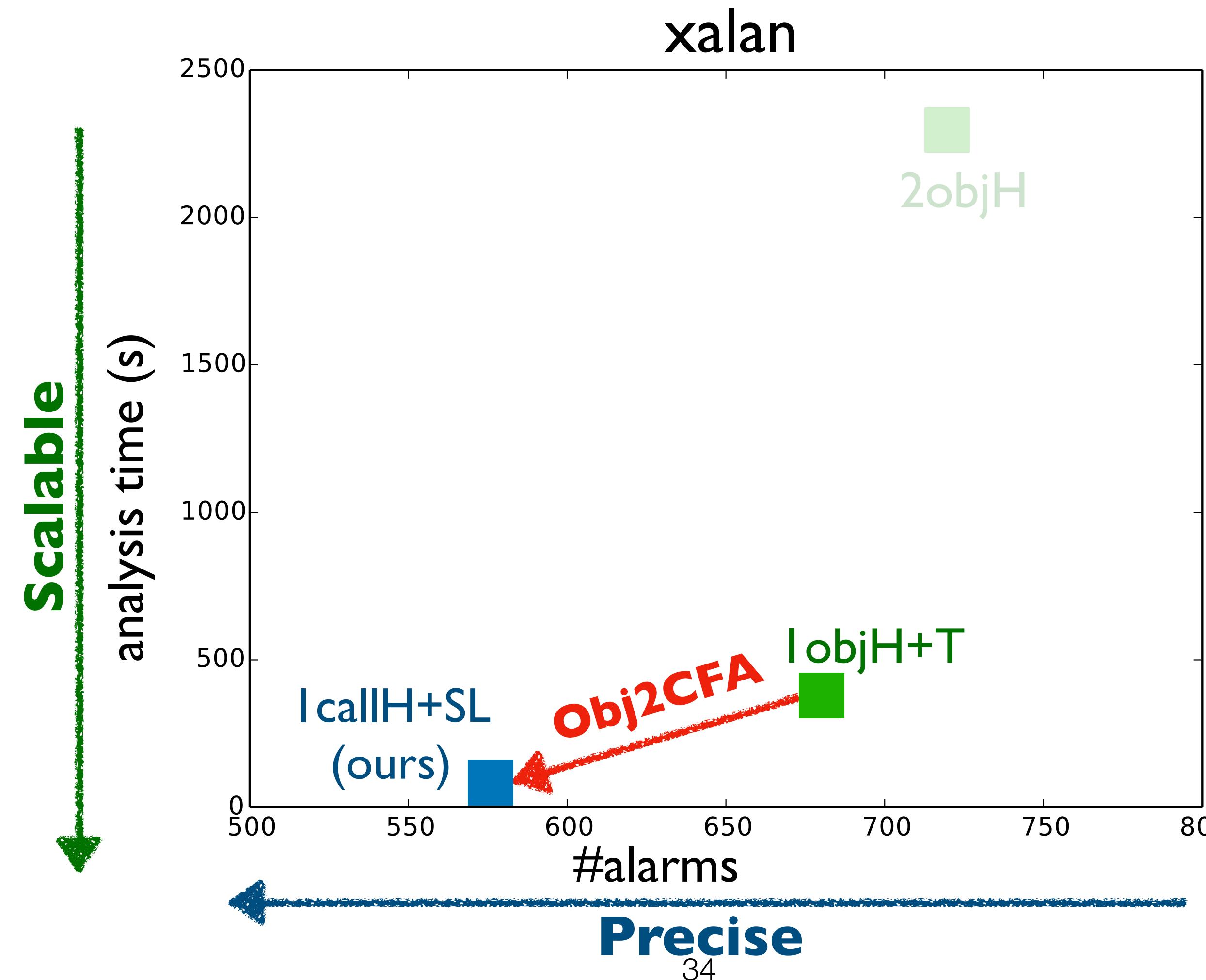
- Limitation of call-site sensitivity is **removed**
- Limitation of object sensitivity is **not removed**

Our claim

If context tunneling is included,
call-site sensitivity is more precise than object sensitivity

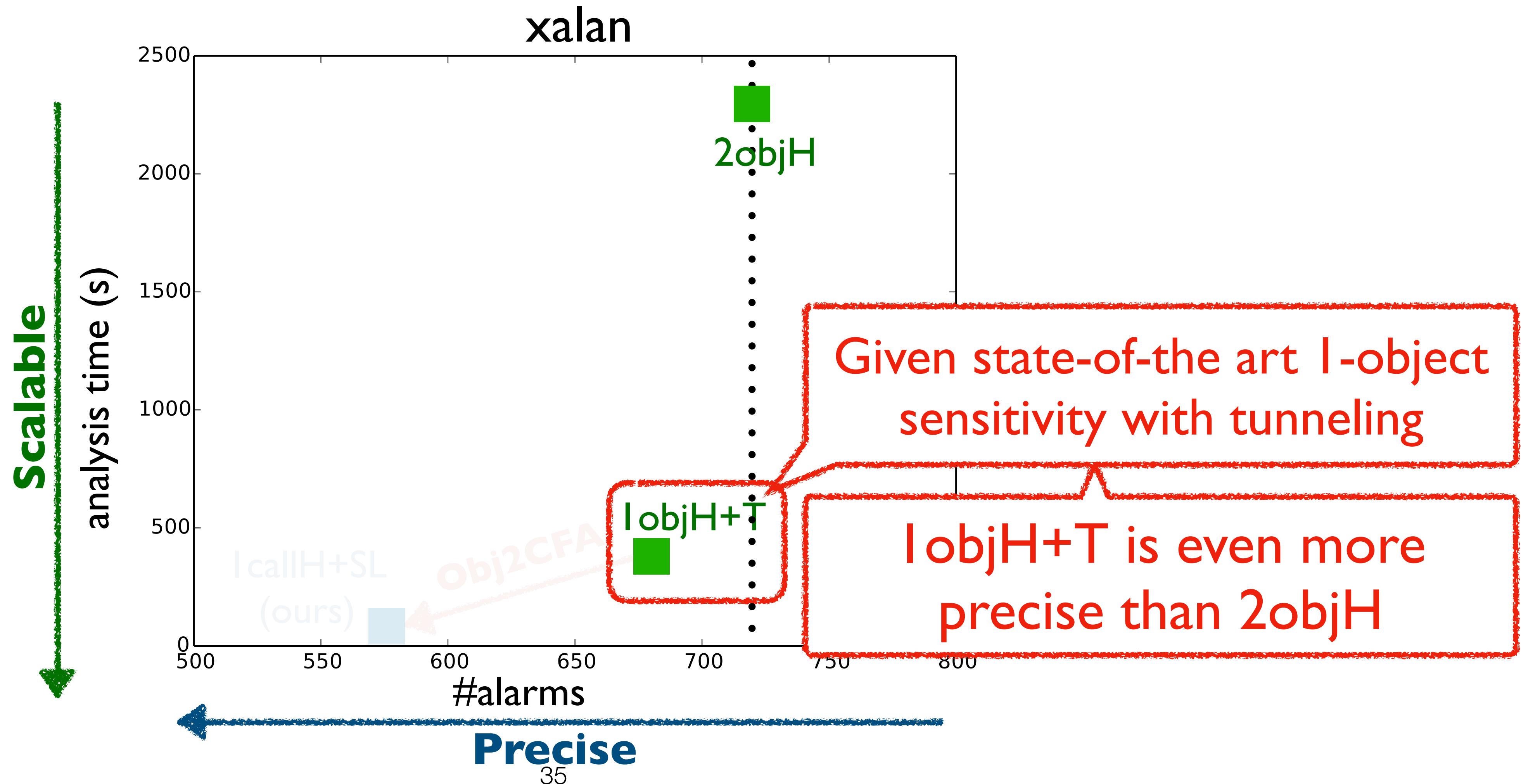
Our Technique : **Obj2CFA**

- **Obj2CFA** transforms a given **object sensitivity** into a more precise **CFA**



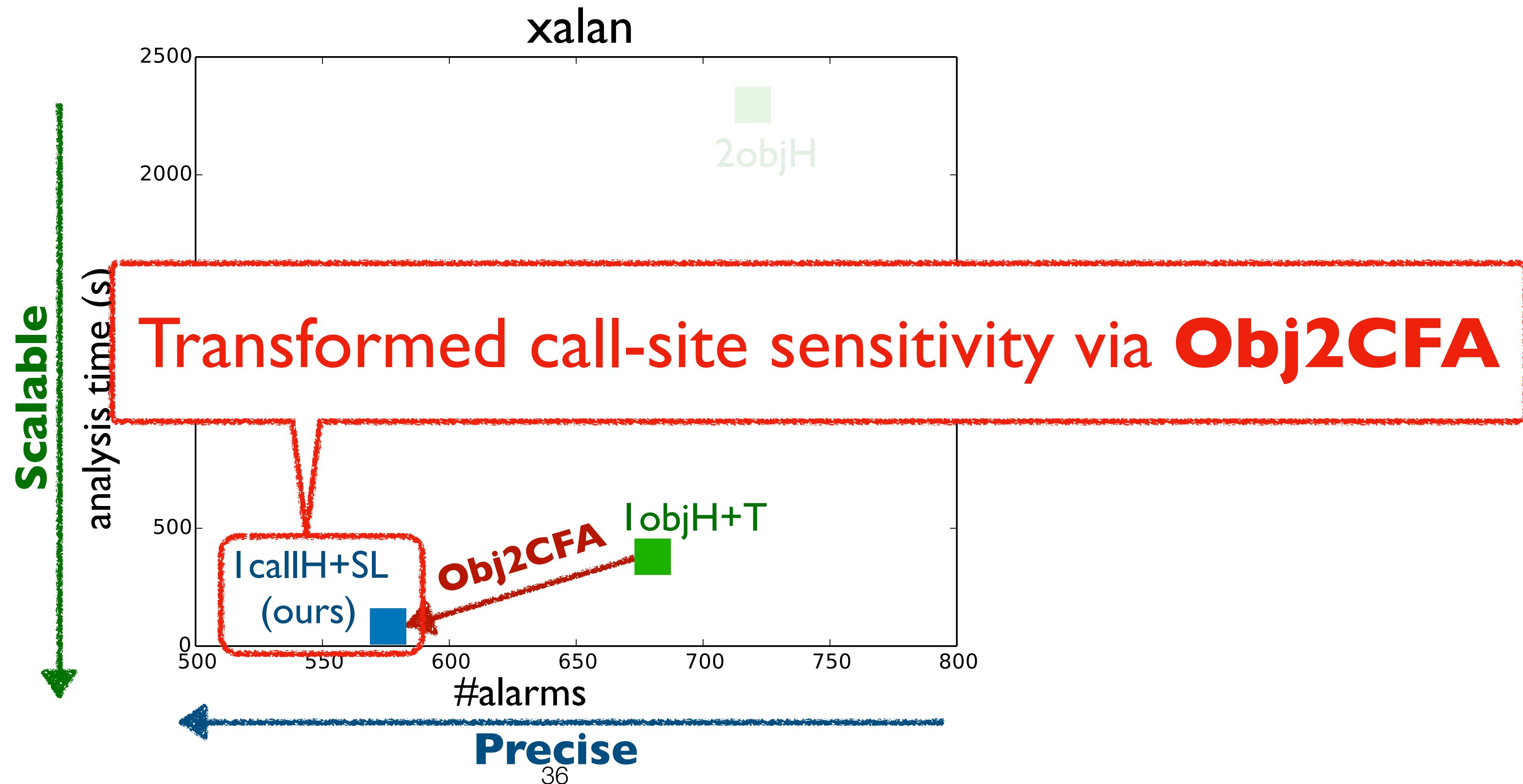
Our Technique : **Obj2CFA**

- **Obj2CFA** transforms a given **object sensitivity** into a more precise **CFA**



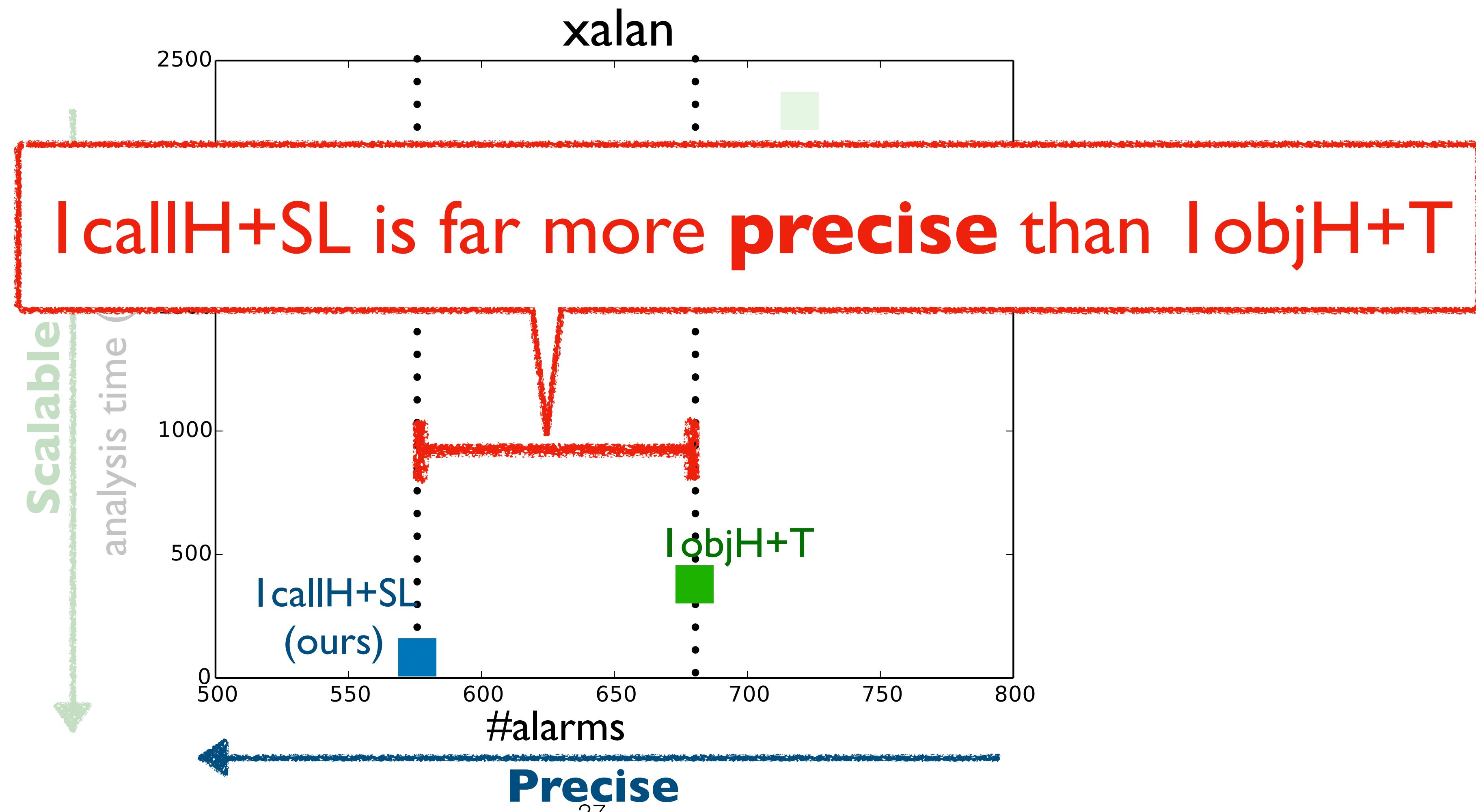
Our Technique : **Obj2CFA**

- **Obj2CFA** transforms a given **object sensitivity** into a more precise **CFA**



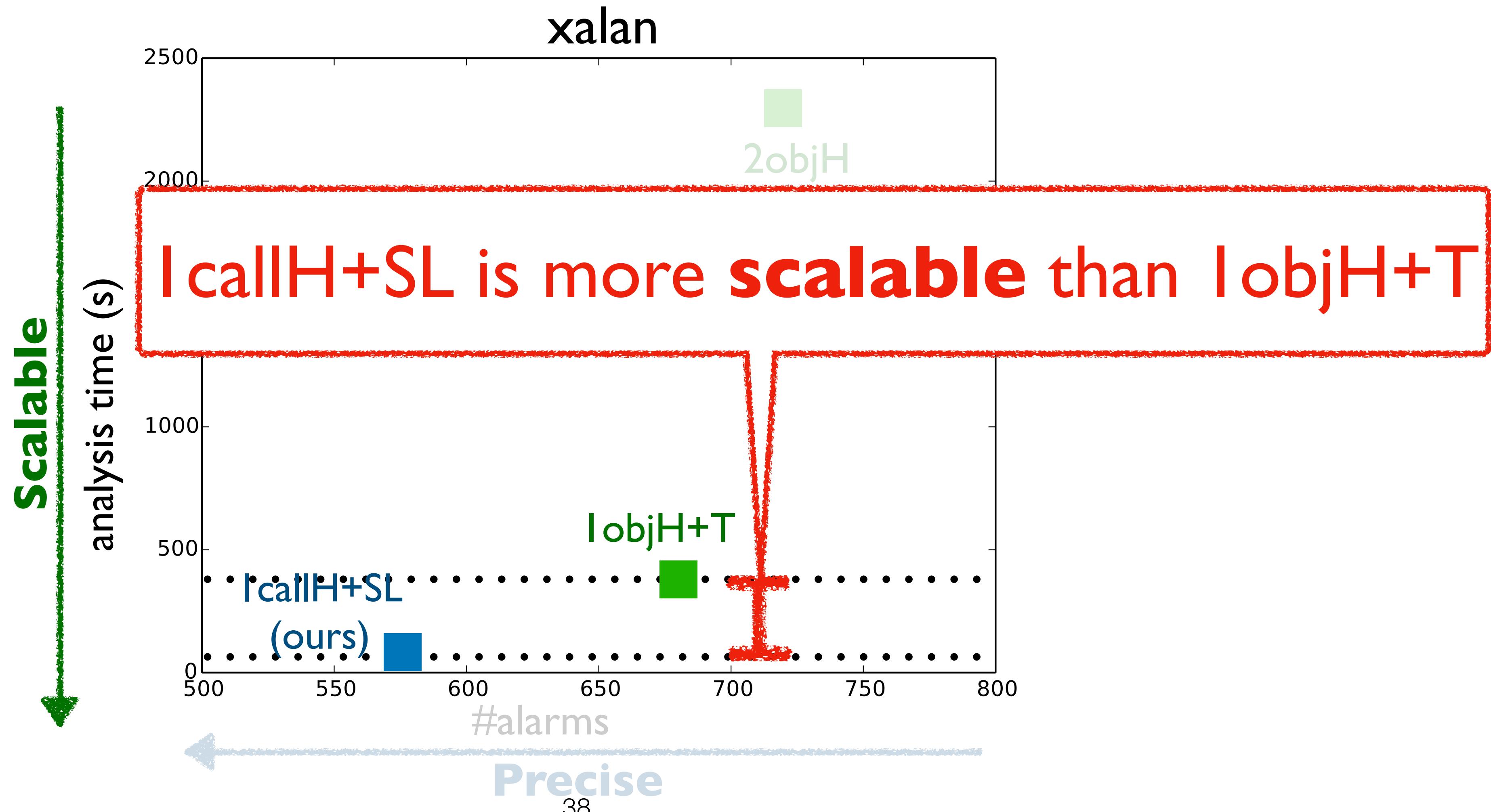
Our Technique : **Obj2CFA**

- **Obj2CFA** transforms a given **object sensitivity** into a more precise **CFA**



Our Technique : **Obj2CFA**

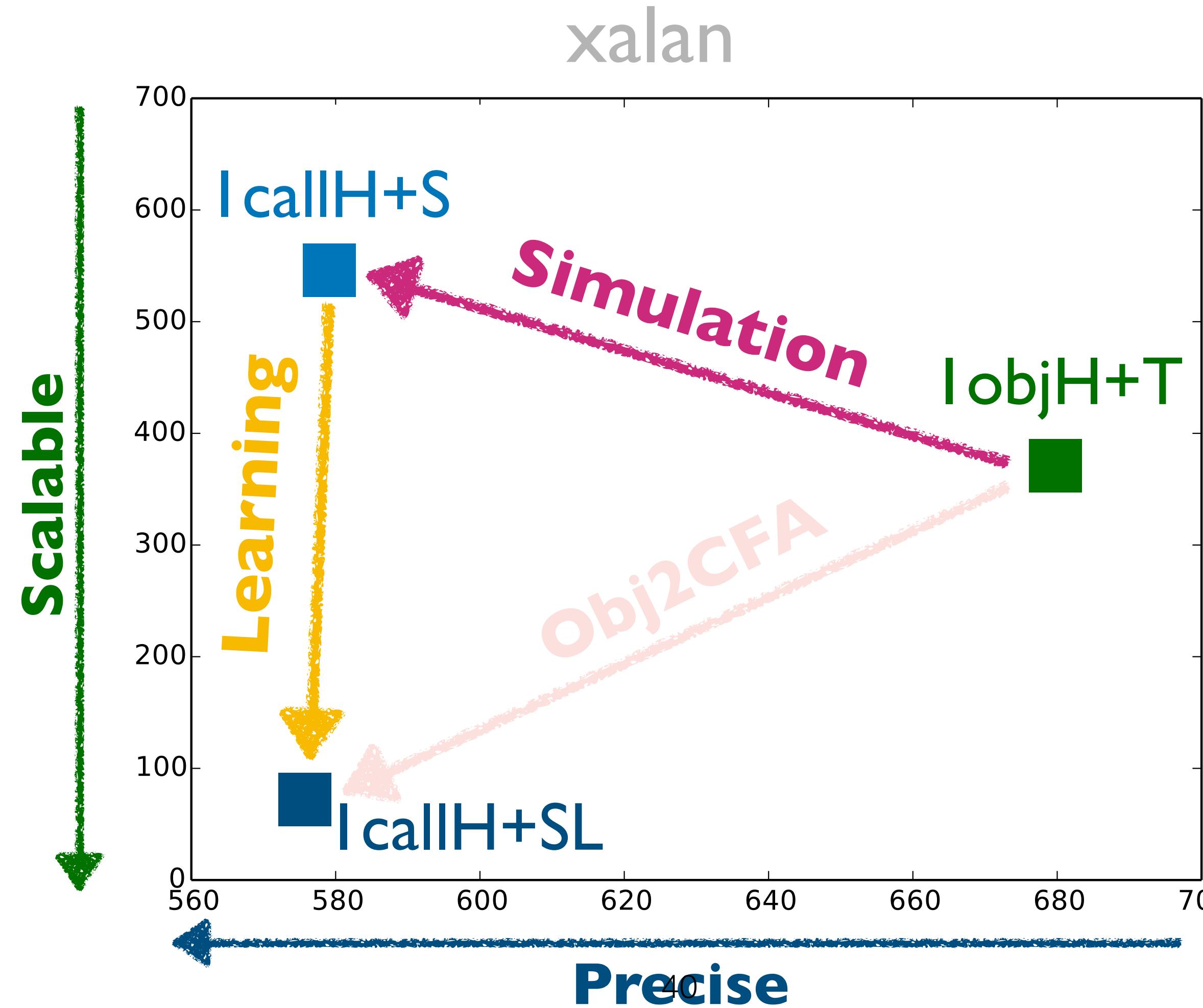
- **Obj2CFA** transforms a given **object sensitivity** into a more precise **CFA**



Detail of Obj2CFA

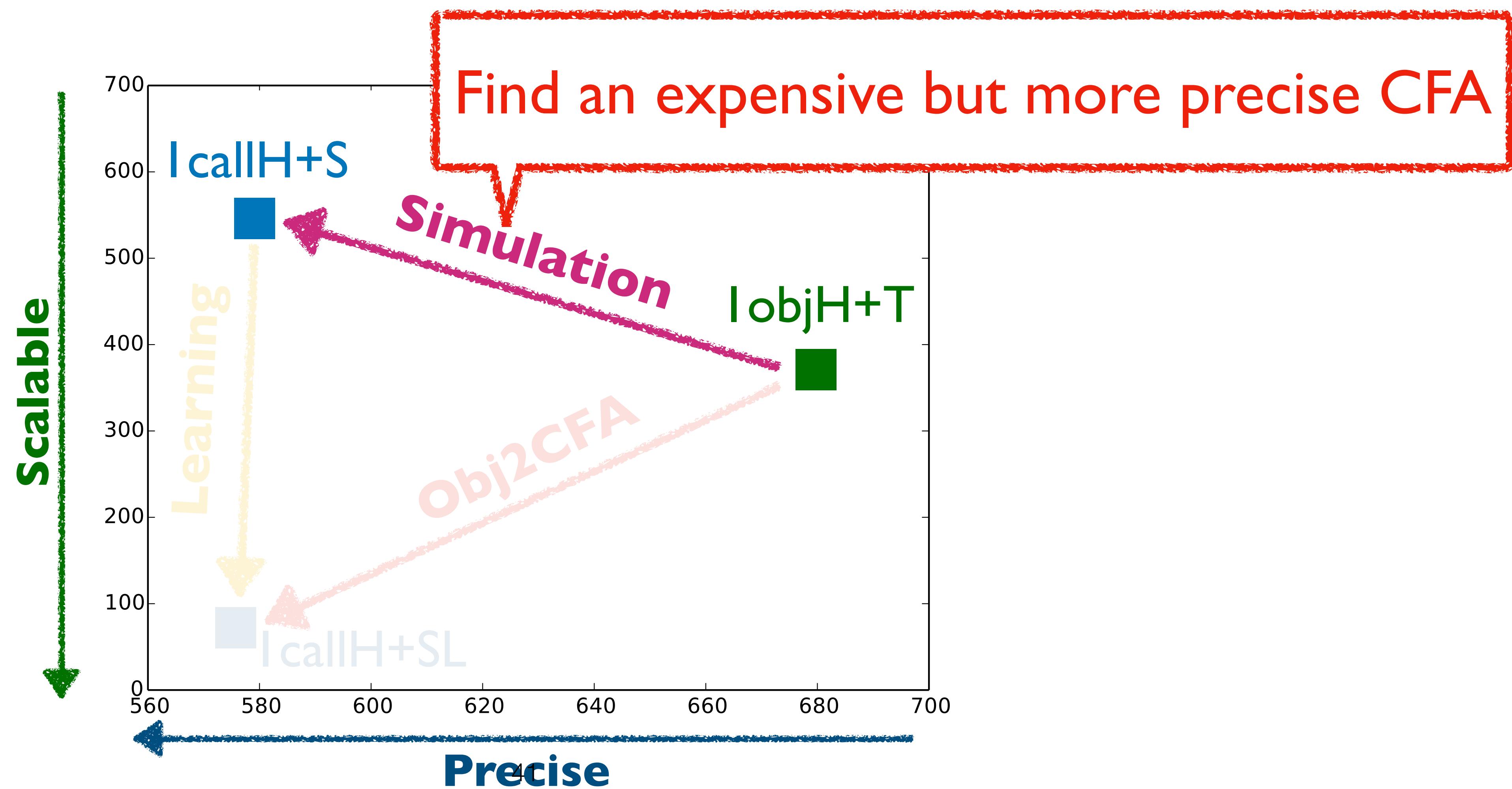
Our Technique : Obj2CFA

- **Obj2CFA** consists of **simulation** and simulation-guided **learning**



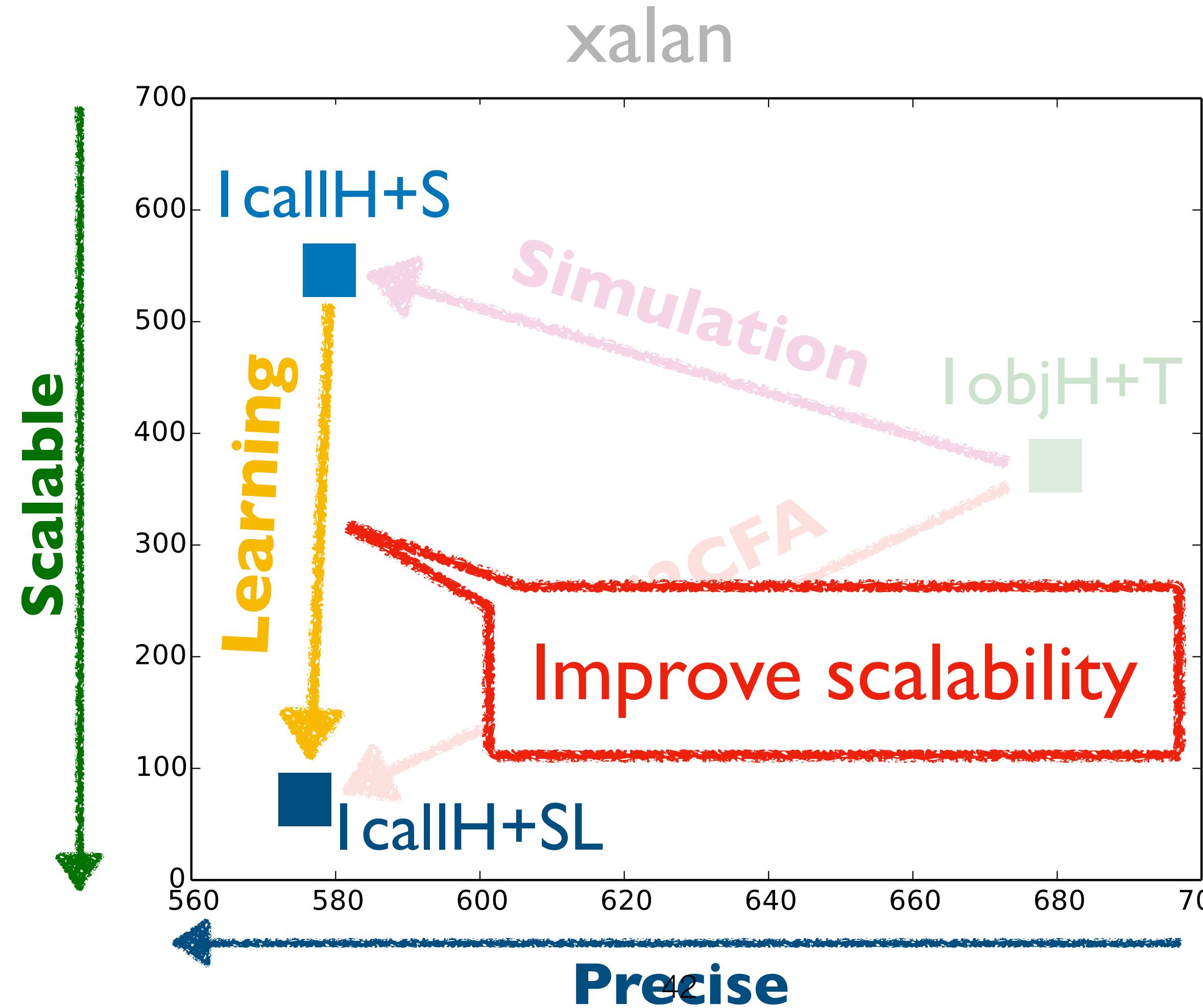
Our Technique : Obj2CFA

- **Obj2CFA** consists of **simulation** and simulation-guided **learning**



Our Technique : Obj2CFA

- **Obj2CFA** consists of **simulation** and simulation-guided **learning**



Technique I: Simulation

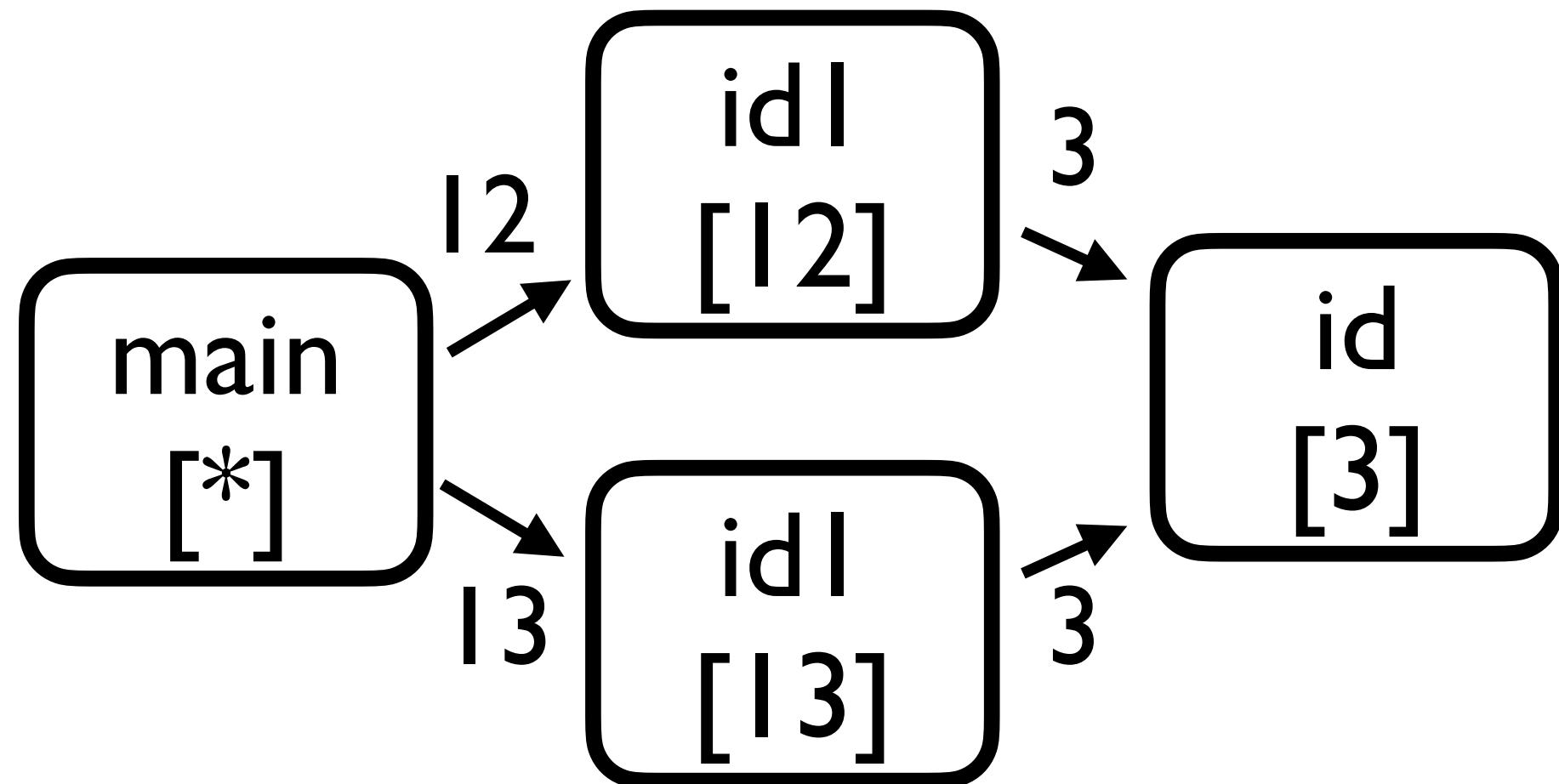
- Running example to illustrate Simulation

```
1: class C{  
2:   id(v){return v;}  
3:   idI(v){return id(v);}  
4:   foo(){  
5:     A a = (A) this.id(new A());}//query1  
6:     B b = (B) this.id(new B());}//query2  
7: }  
8: main(){  
9:   c1 = new C();//C1  
10:  c2 = new C();//C2  
11:  c3 = new C();//C3  
12:  A a = (A) c1.idI(new A());//query3  
13:  B b = (B) c2.idI(new B());//query4  
14:  c3.foo();  
15: }
```

Technique I: Simulation

- Running example to illustrate Simulation

```
1: class C{  
2:     id(v){return v;}  
3:     idI(v){return id(v);} idI(v){return id(v);} → Limitation of conventional I-CFA  
4:     foo(){  
5:         A a = (A) this.id(new A());//query1  
6:         B b = (B) this.id(new B());//query2  
7:     }  
8:     main(){  
9:         c1 = new C();//C1  
10:        c2 = new C();//C2  
11:        c3 = new C();//C3  
12:        A a = (A) c1.idI(new A());//query3 A a = (A) c1.idI(new A());//query3  
13:        B b = (B) c2.idI(new B());//query4 B b = (B) c2.idI(new B());//query4  
14:        c3.foo();  
15:    }
```



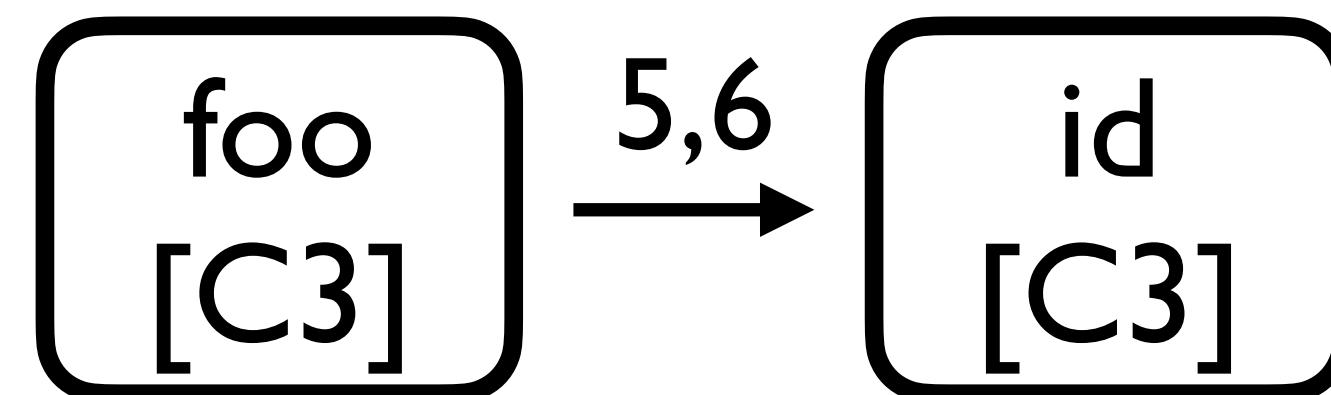
Technique I: Simulation

- Running example to illustrate Simulation

```
1: class C{  
2:   id(v){return v;}  
3:   idI(v){return id(v);}  
4:   foo(){  
5:     A a = (A) this.id(new A());}//query1  
6:     B b = (B) this.id(new B());}//query2  
7: }  
8: main(){  
9:   c1 = new C();//C1  
10:  c2 = new C();//C2  
11:  c3 = new C();//C3  
12:  A a = (A) c1.idI(new A());//query3  
13:  B b = (B) c2.idI(new B());//query4  
14:  c3.foo();  
15: }
```



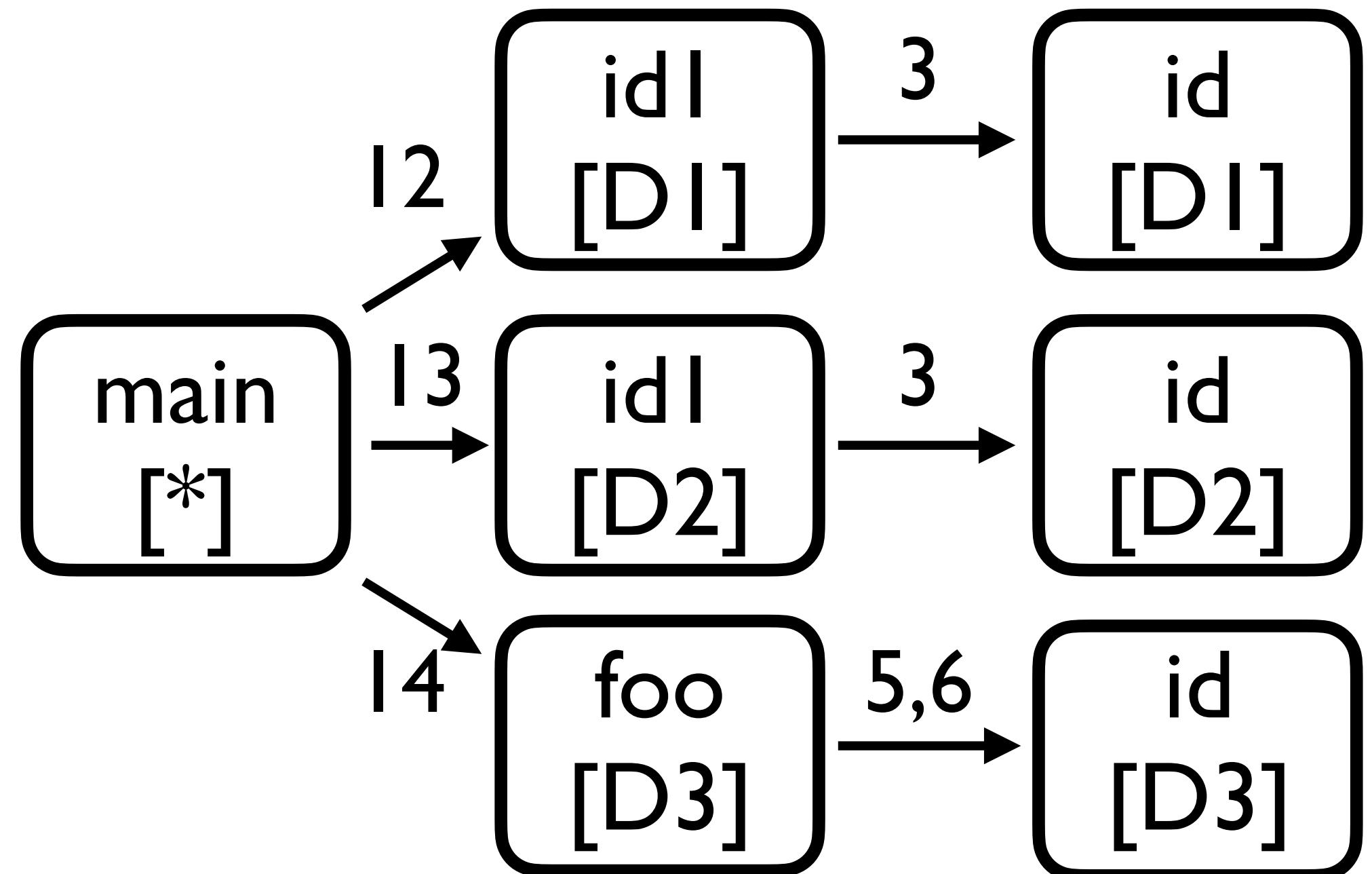
Limitation of object sensitivity



Technique I: Simulation

- Given **object sensitivity** is conventional 1-object sensitivity (e.g., $T = \emptyset$)

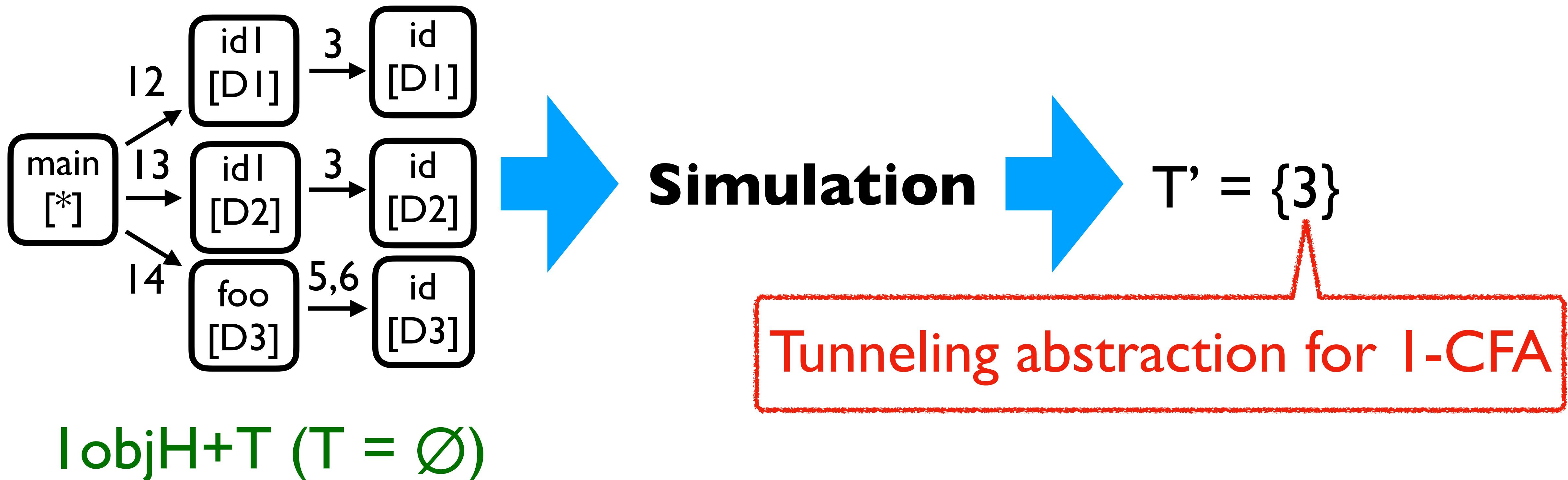
```
I: class C{  
2:   id(v){return v;}  
3:   idI(v){return id(v);}  
4:   foo(){  
5:     A a = (A) this.id(new A());}//query1  
6:     B b = (B) this.id(new B());}//query2  
7: }  
8: main(){  
9:   c1 = new C();//C1  
10:  c2 = new C();//C2  
11:  c3 = new C();//C3  
12:  A a = (A) c1.idI(new A());//query3  
13:  B b = (B) c2.idI(new B());//query4  
14:  c3.foo();  
15: }
```



IobjH+T ($T = \emptyset$)

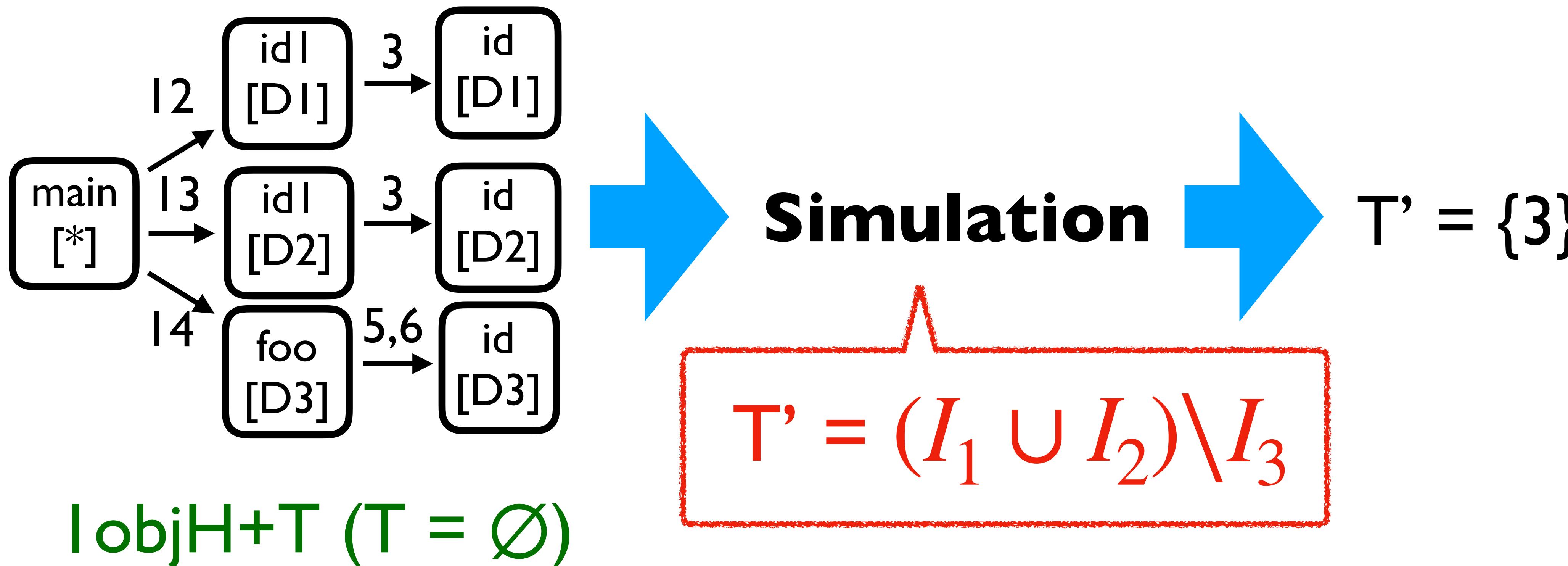
Technique I: Simulation

- **Simulation** takes a call-graph and produce a tunneling abstraction for CFA



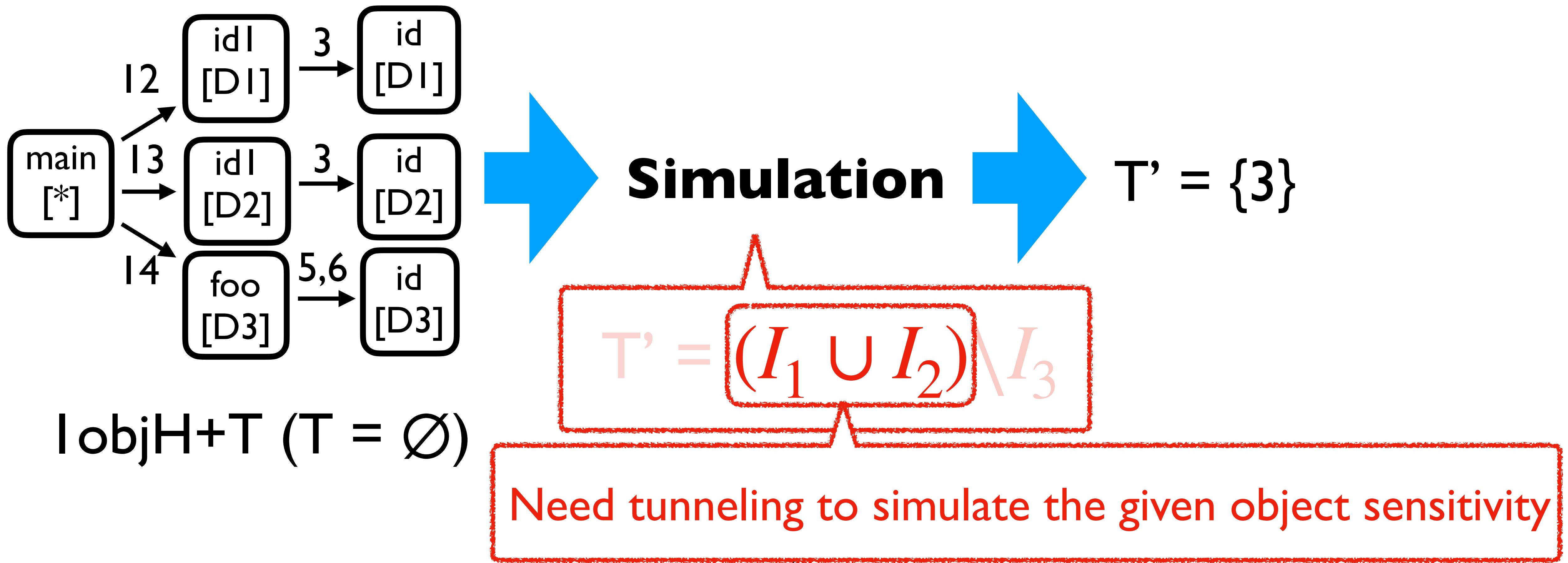
Technique I: Simulation

- **Simulation** takes a call-graph and produce a tunneling abstraction for CFA



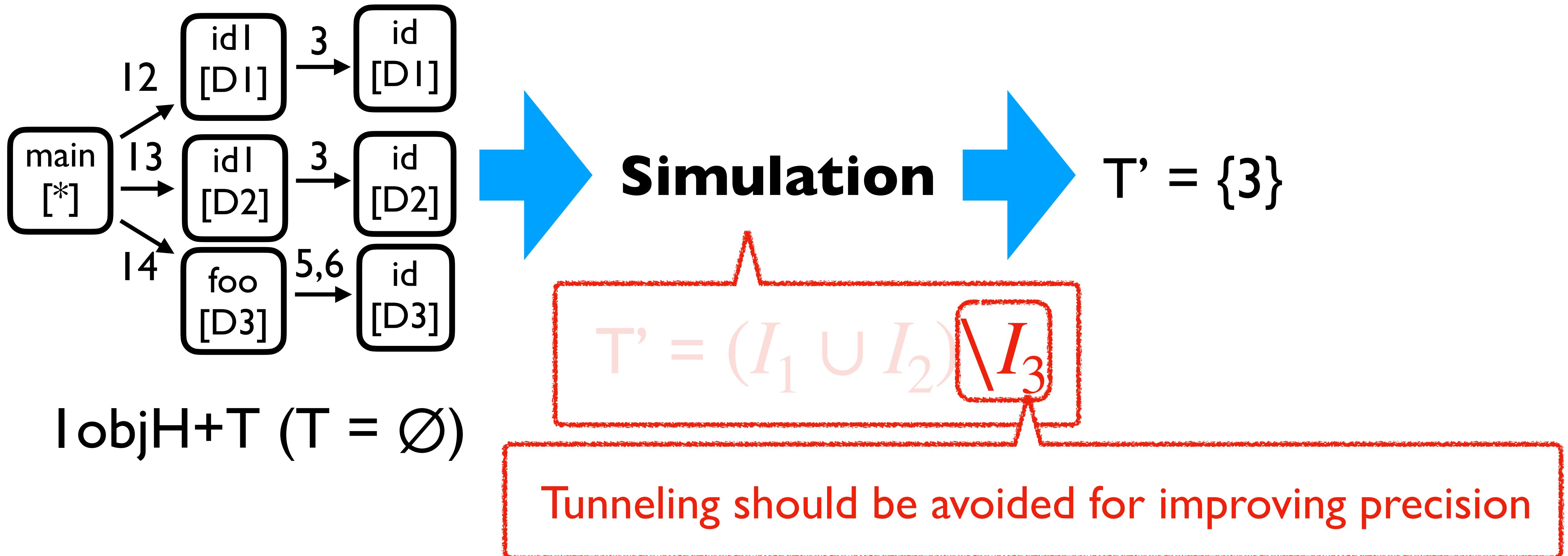
Technique I: Simulation

- **Simulation** takes a call-graph and produce a tunneling abstraction for CFA



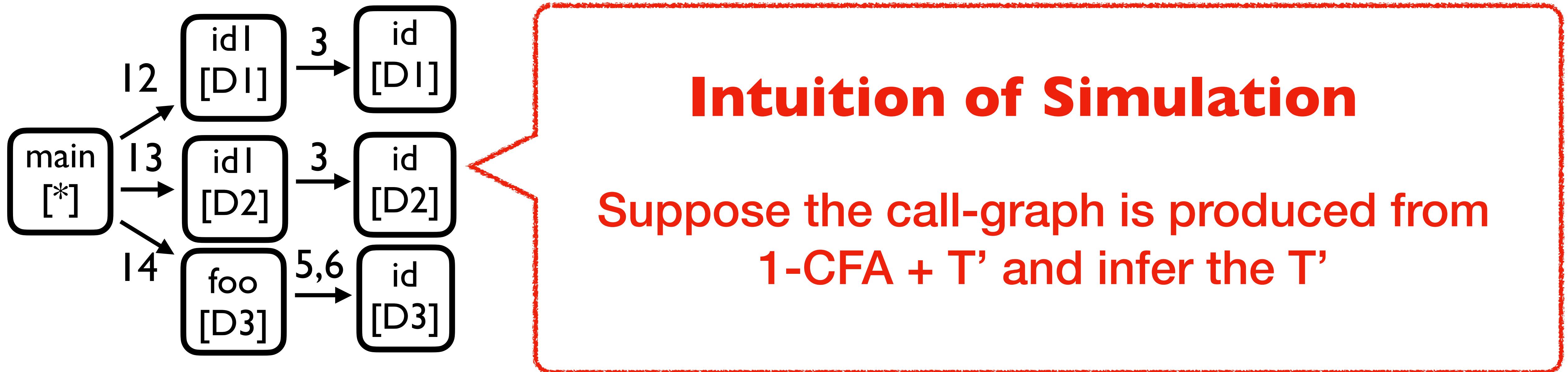
Technique I: Simulation

- **Simulation** takes a call-graph and produce a tunneling abstraction for CFA



Technique I: Simulation

- **Simulation** takes a call-graph and produce a tunneling abstraction for CFA



~~IobjH+T (T = \emptyset)~~

IcallH+T'

What is T'?

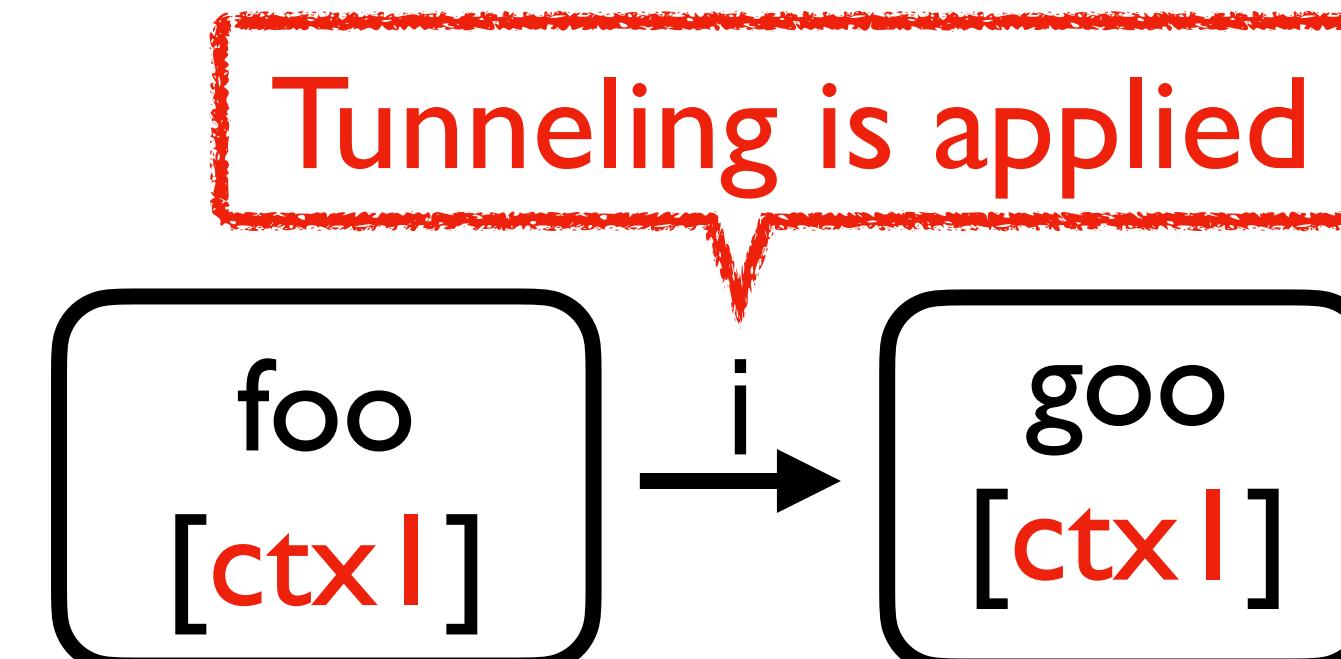
Intuition Behind Simulation ($I_1 \cup I_2$)

- If tunneling is applied to i , two properties inevitably appear at i

We track the two properties to find the T'

Intuition Behind Simulation ($I_1 \cup I_2$)

- If tunneling is applied to i, two properties inevitably appear at i



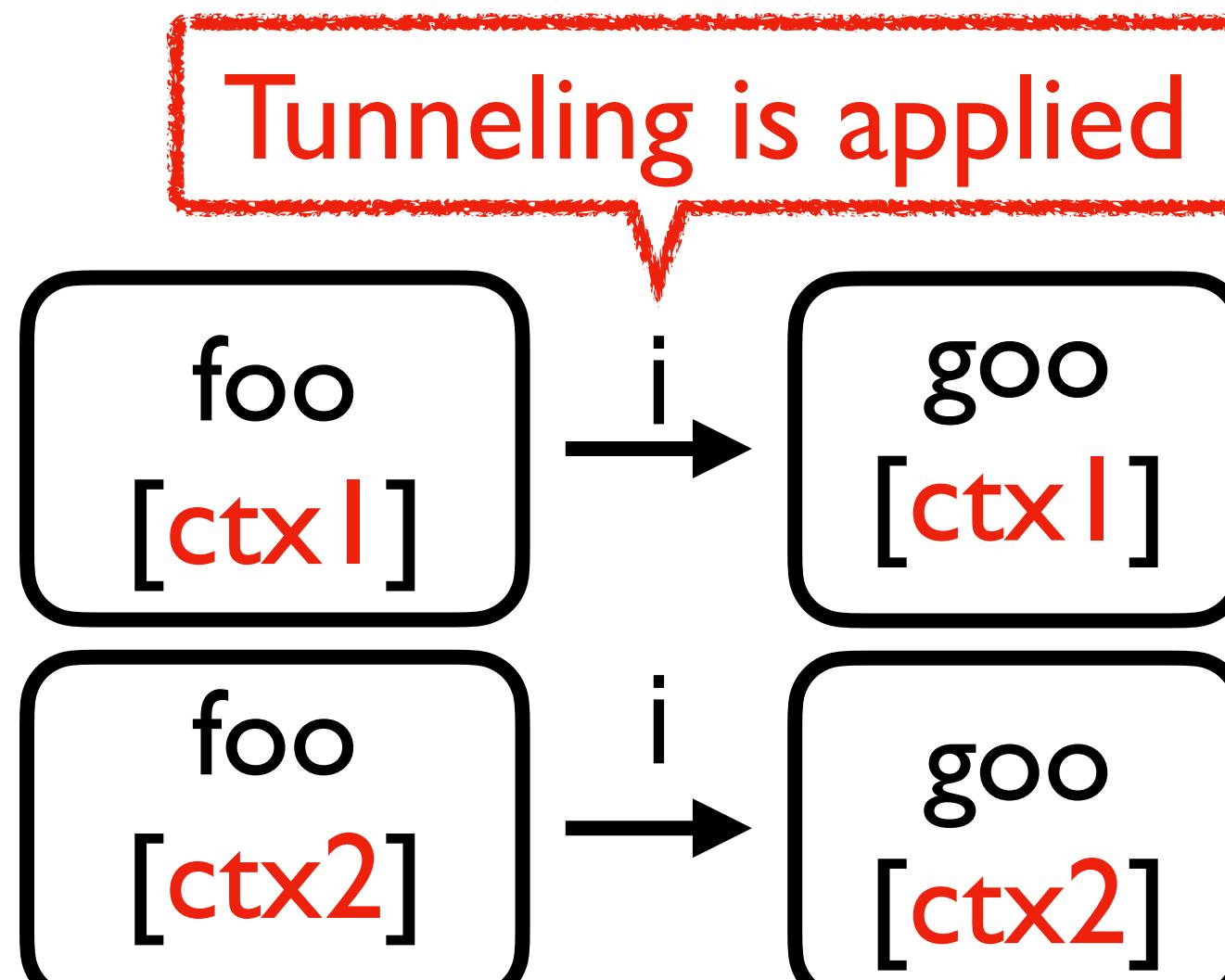
Property of context tunneled call-sites

- Property I: caller and callee methods have the **same context**

I_1

Intuition Behind Simulation ($I_1 \cup I_2$)

- If tunneling is applied to i, two properties inevitably appear at i



Property of context tunneled invocations

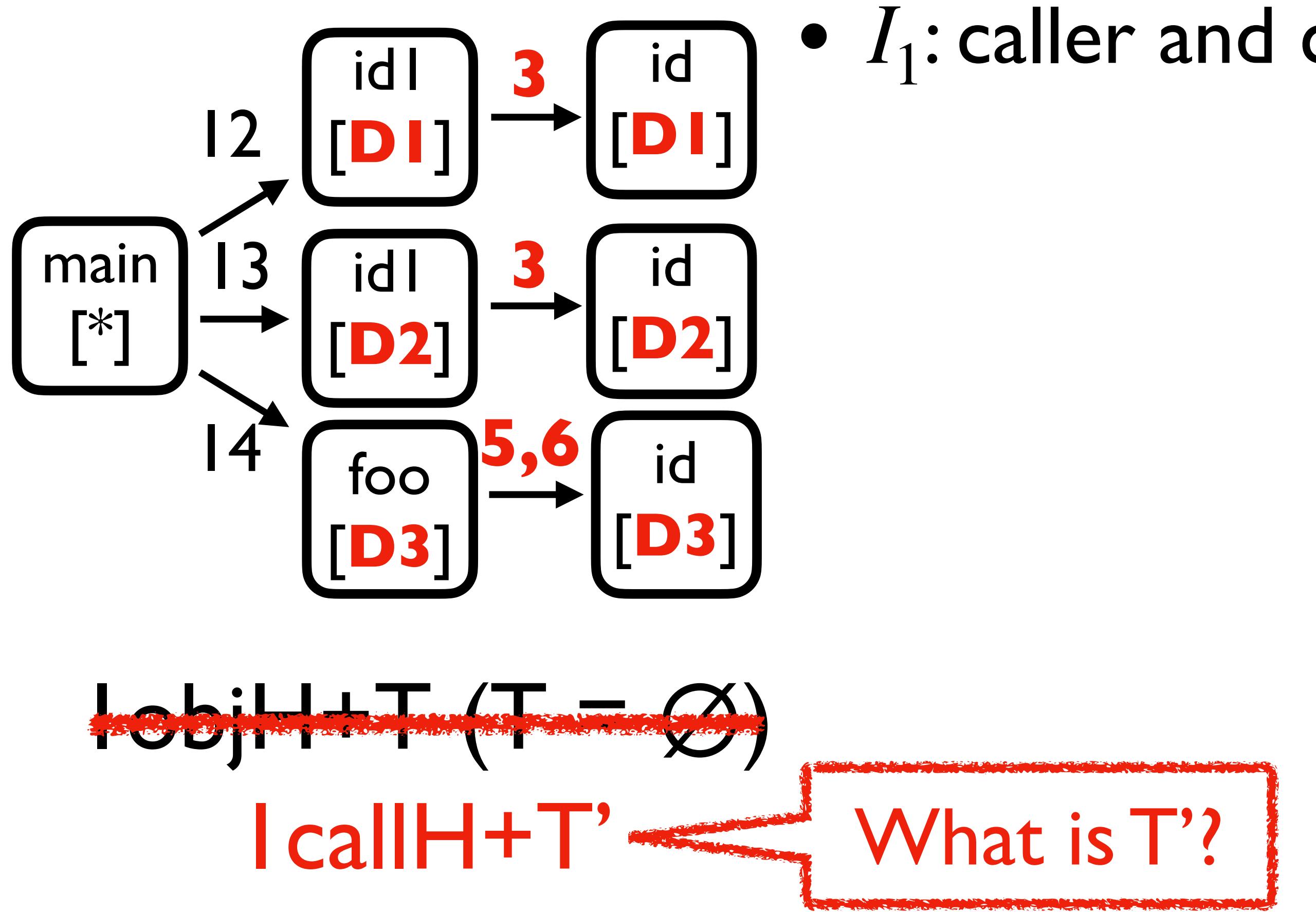
- Property 2: different caller contexts imply different callee contexts

I_1

I_2

Intuition Behind Simulation ($I_1 \cup I_2$)

- Suppose given call-graph is produced from $\text{IcallH+T}'$ and infer what T' is

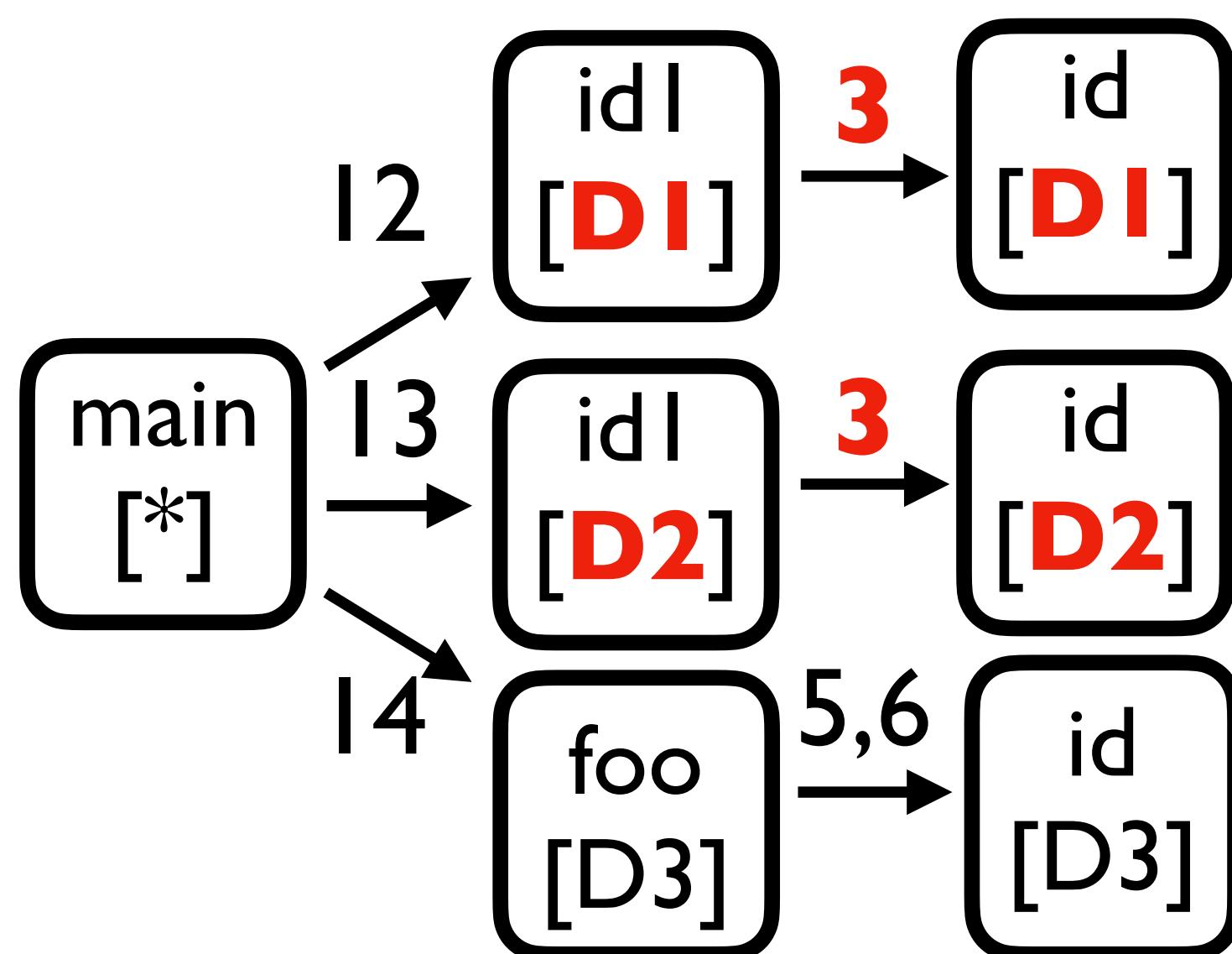


- I_1 : caller and callee methods have the **same context**

$$I_1 = \{3, 5, 6\}$$

Intuition Behind Simulation ($I_1 \cup I_2$)

- Suppose given call-graph is produced from IcallH+T' and infer what T' is



- I_1 : caller and callee methods have the **same context**

$$I_1 = \{3, 5, 6\}$$

- I_2 : different caller ctx imply different callee ctx

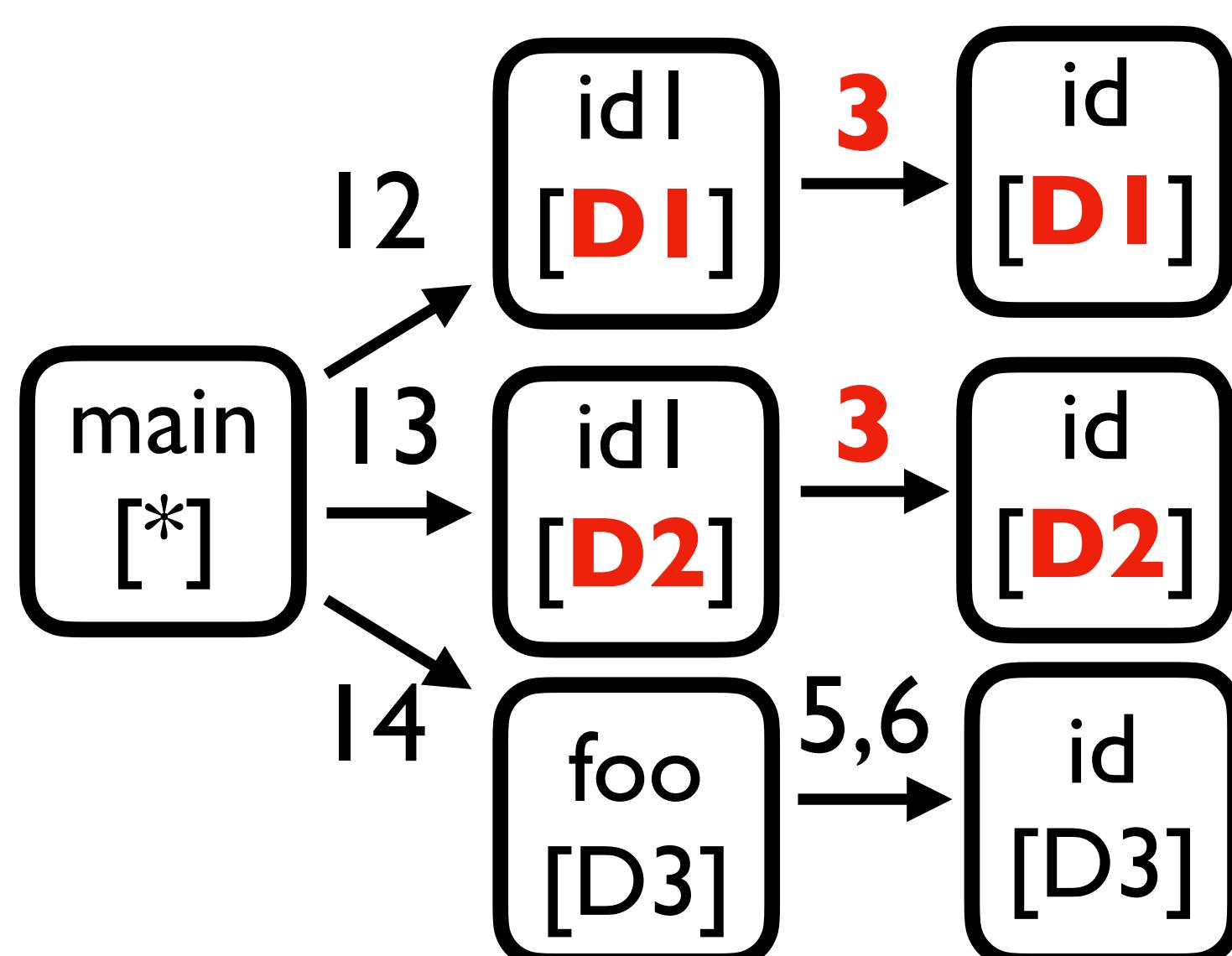
$$I_2 = \{3\}$$

~~$\text{IobjH+T} (T = \emptyset)$~~

IcallH+T'  What is T' ?

Intuition Behind Simulation ($I_1 \cup I_2$)

- Suppose given call-graph is produced from IcallH+T' and infer what T' is



- I_1 : caller and callee methods have the **same context**

$$I_1 = \{3, 5, 6\}$$

- I_2 : different caller ctx imply different callee ctx

$$I_2 = \{3\}$$

~~$\text{IobjH+T} (T = \emptyset)$~~

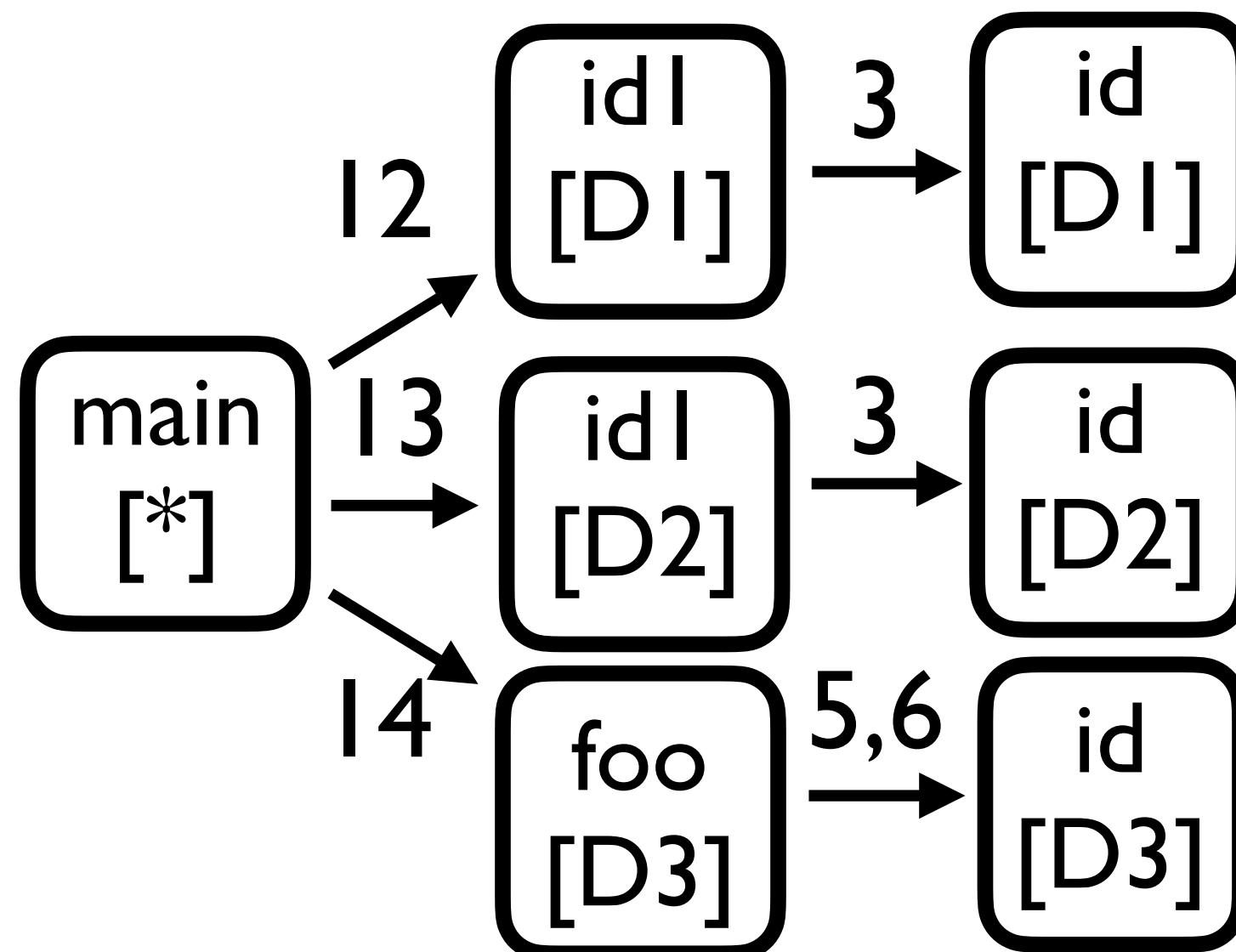
IcallH+T'

What is T' ?

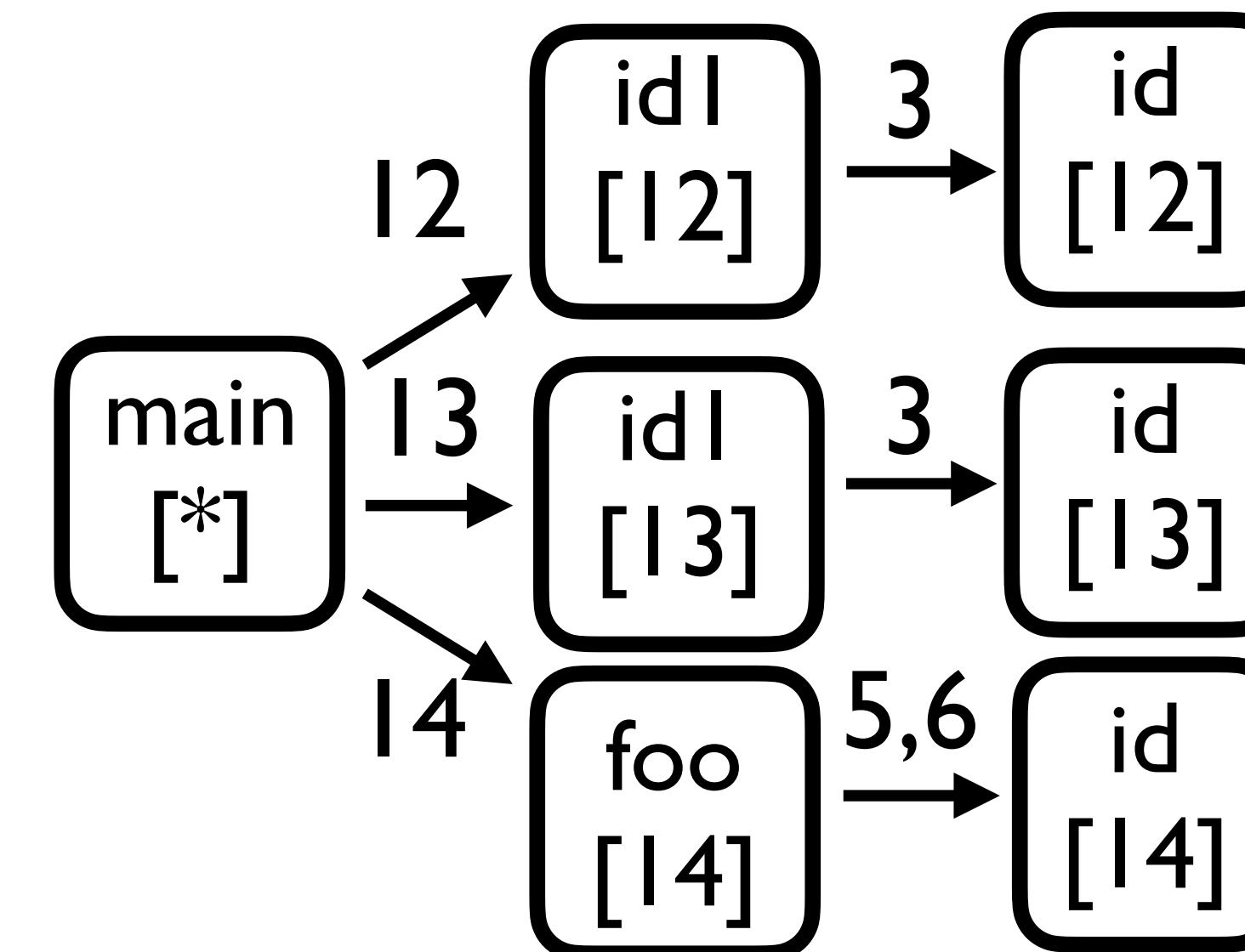
$$T' = I_1 \cup I_2 = \{3, 5, 6\}$$

Intuition Behind Simulation ($I_1 \cup I_2$)

- Suppose given call-graph is produced from $I_{\text{call}}H+T'$ and infer what T' is



$I_{\text{obj}}H+T$ ($T = \emptyset$)

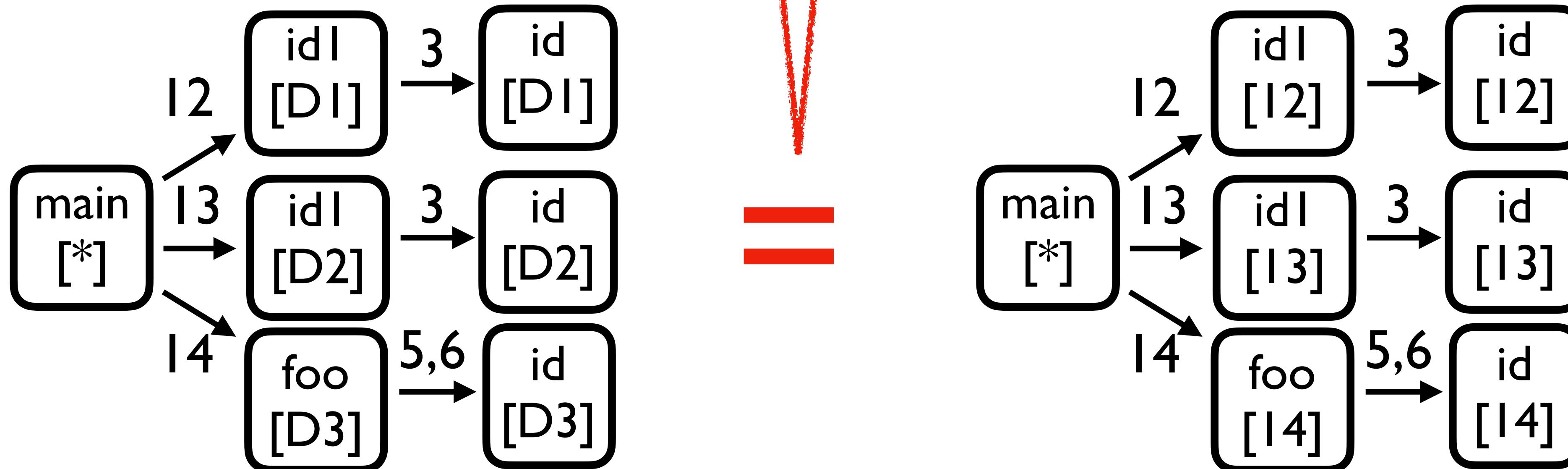


$I_{\text{call}}H+T'$ ($T' = \{3,5,6\}$)

Intuition Behind Simulation ($I_1 \cup I_2$)

- Suppose given call-graph and infer what T' is

Exactly the same analyses

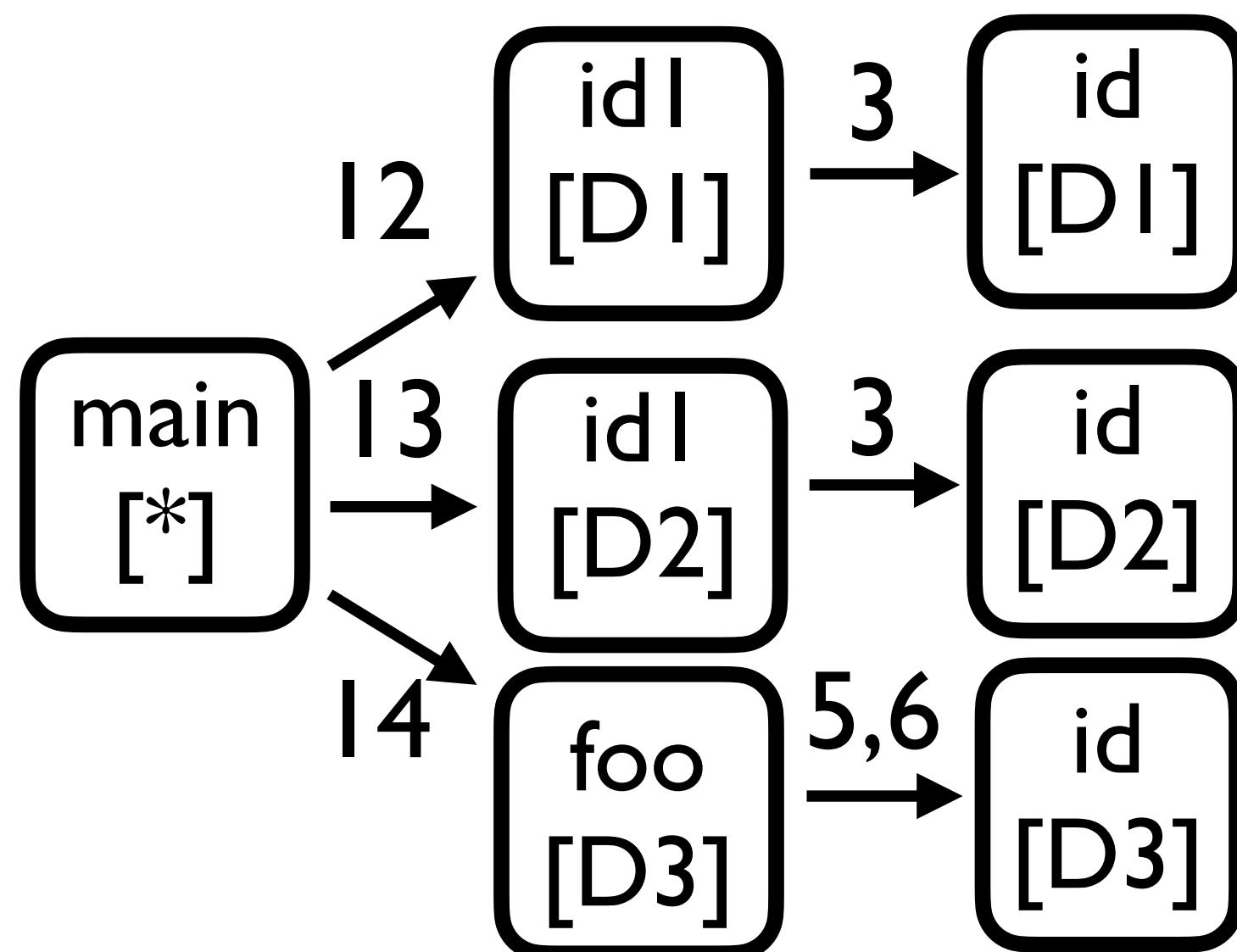


$I_{\text{objH}} + T$ ($T = \emptyset$)

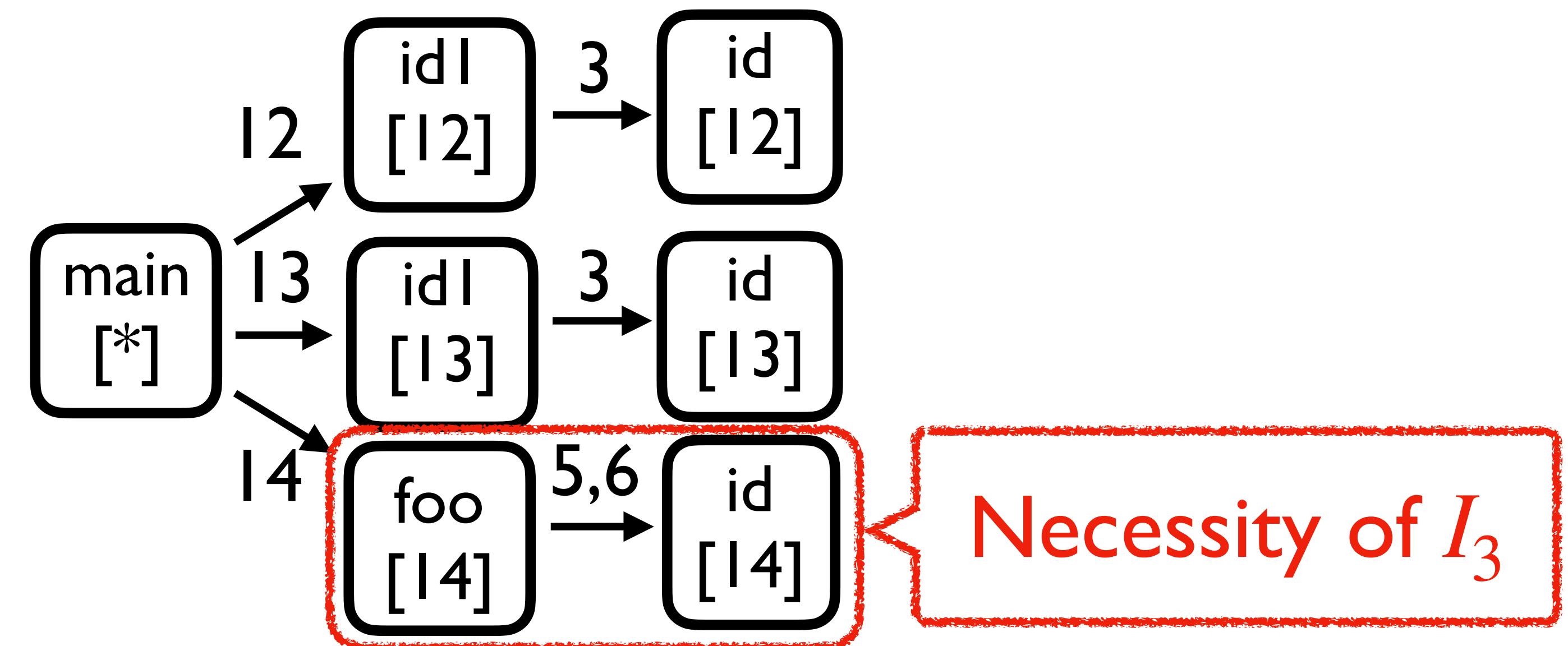
$I_{\text{callH}} + T'$ ($T' = \{3,5,6\}$)

Intuition Behind Simulation ($I_1 \cup I_2$)

- Suppose given call-graph is produced from $I_{\text{callH+T'}}$ and infer what T' is



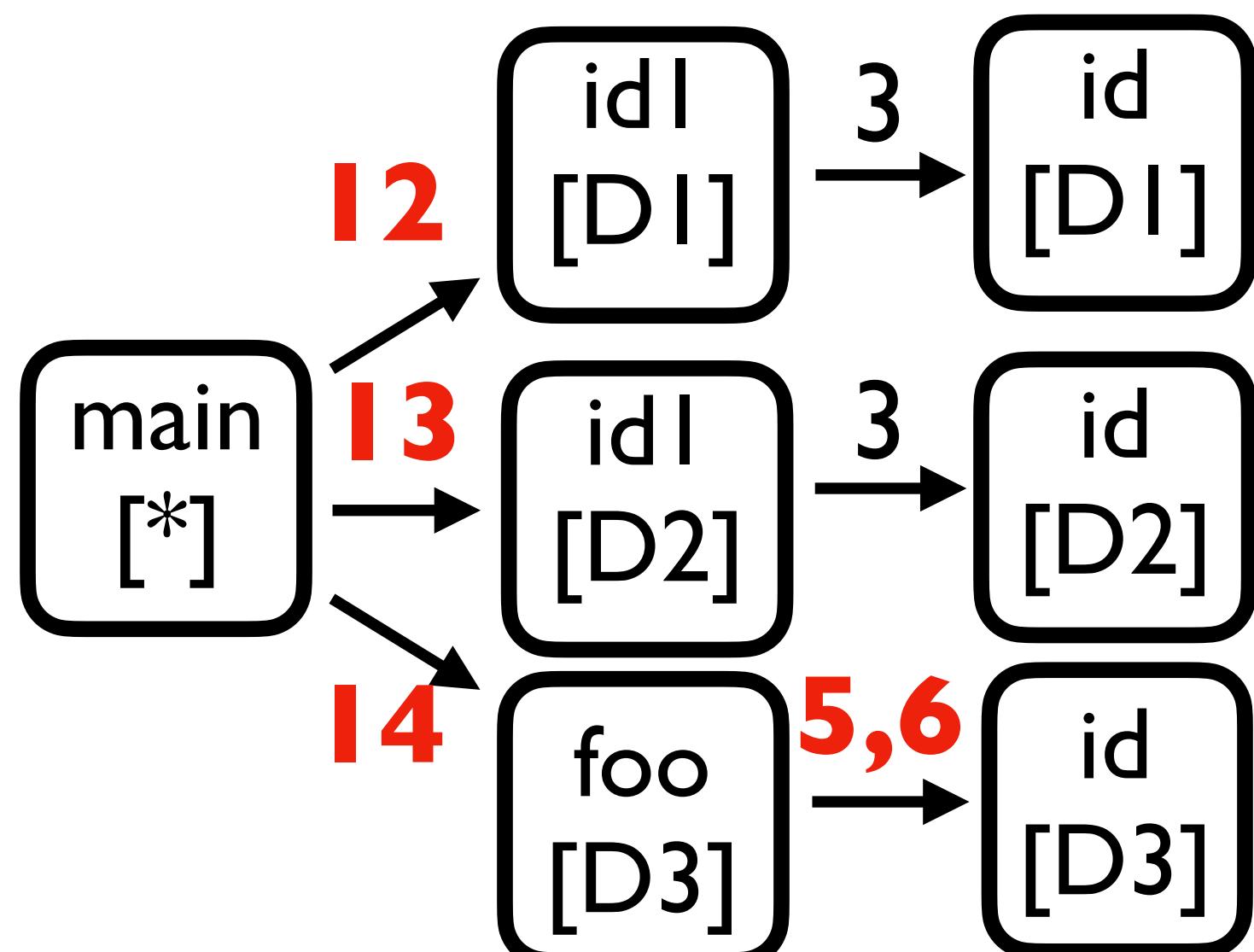
$I_{\text{objH+T}} (T = \emptyset)$



$I_{\text{callH+T'}} (T' = \{3,5,6\})$

Intuition Behind Simulation (I_3)

- I_3 : Tunneling should be avoided for improving precision



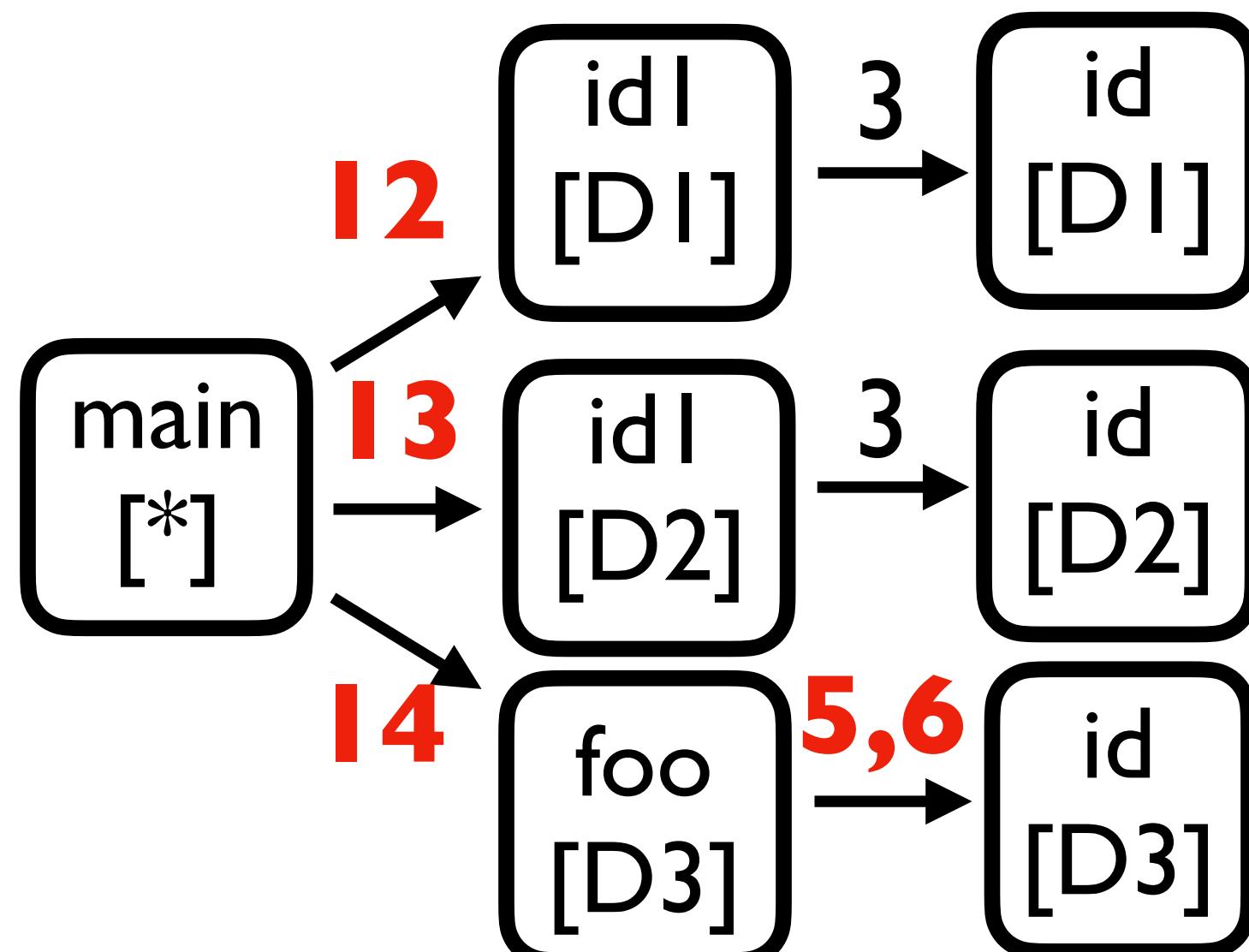
- I_1 : caller and callee methods have the **same context**
 $I_1 = \{3, 5, 6\}$
- I_2 : different caller ctx imply different callee ctx
 $I_2 = \{3\}$
- I_3 : given object sensitivity produced only one context

$\text{lobjH+T} (T = \emptyset)$

$$I_3 = \{5, 6, I2, I3, I4\}$$

Intuition Behind Simulation

- The inferred tunneling abstraction T' is a singleton set $\{3\}$



- I_1 : caller and callee methods have the **same context**
- I_2 : different caller ctx imply different tunneling abstractions
- I_3 : given object sensitivity produced only one context

$\text{lobjH} + T \ (T = \emptyset)$

$$I_3 = \{5, 6, I2, I3, I4\}$$

$$I_1 = \{3, 5, 6\}$$

$$I_2 = \{3\}$$

$$T' = (I_1 \cup I_2) \setminus I_3 = \{3\}$$

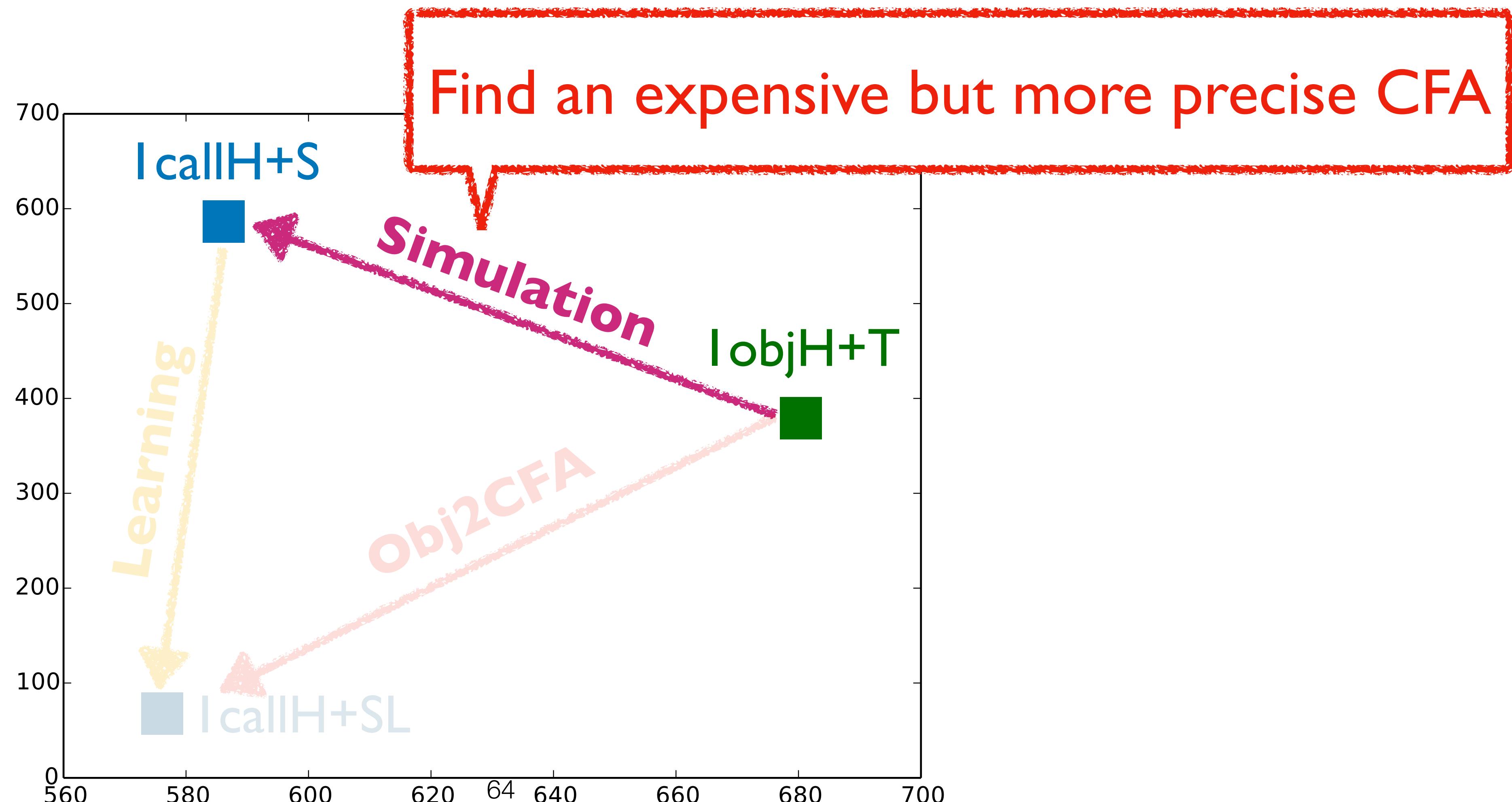
Technique I: Simulation

- With T' , CFA becomes more precise than the given object sensitivity



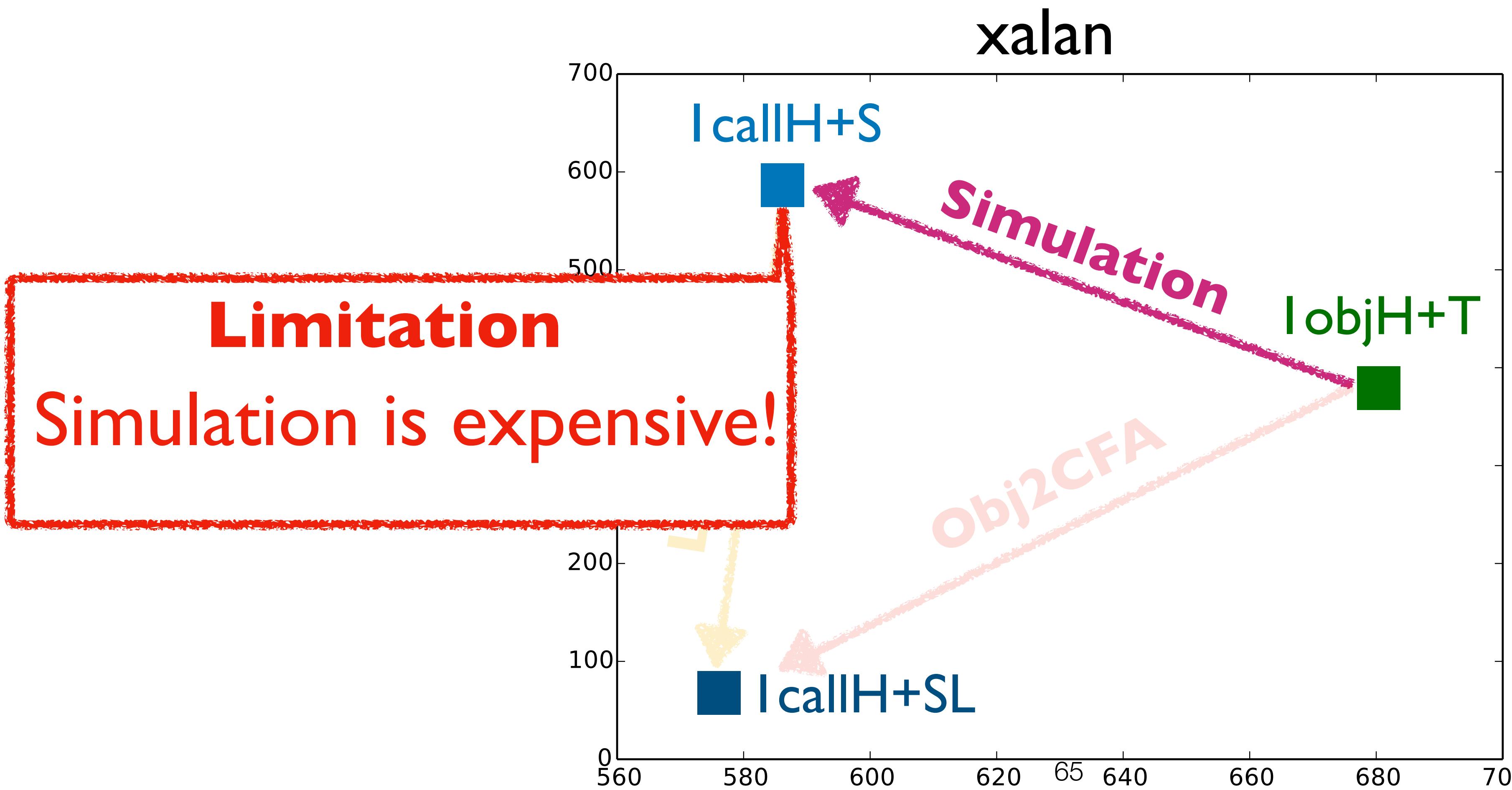
Our Technique : Obj2CFA

- **Obj2CFA** consists of **simulation** and simulation-guided **learning**



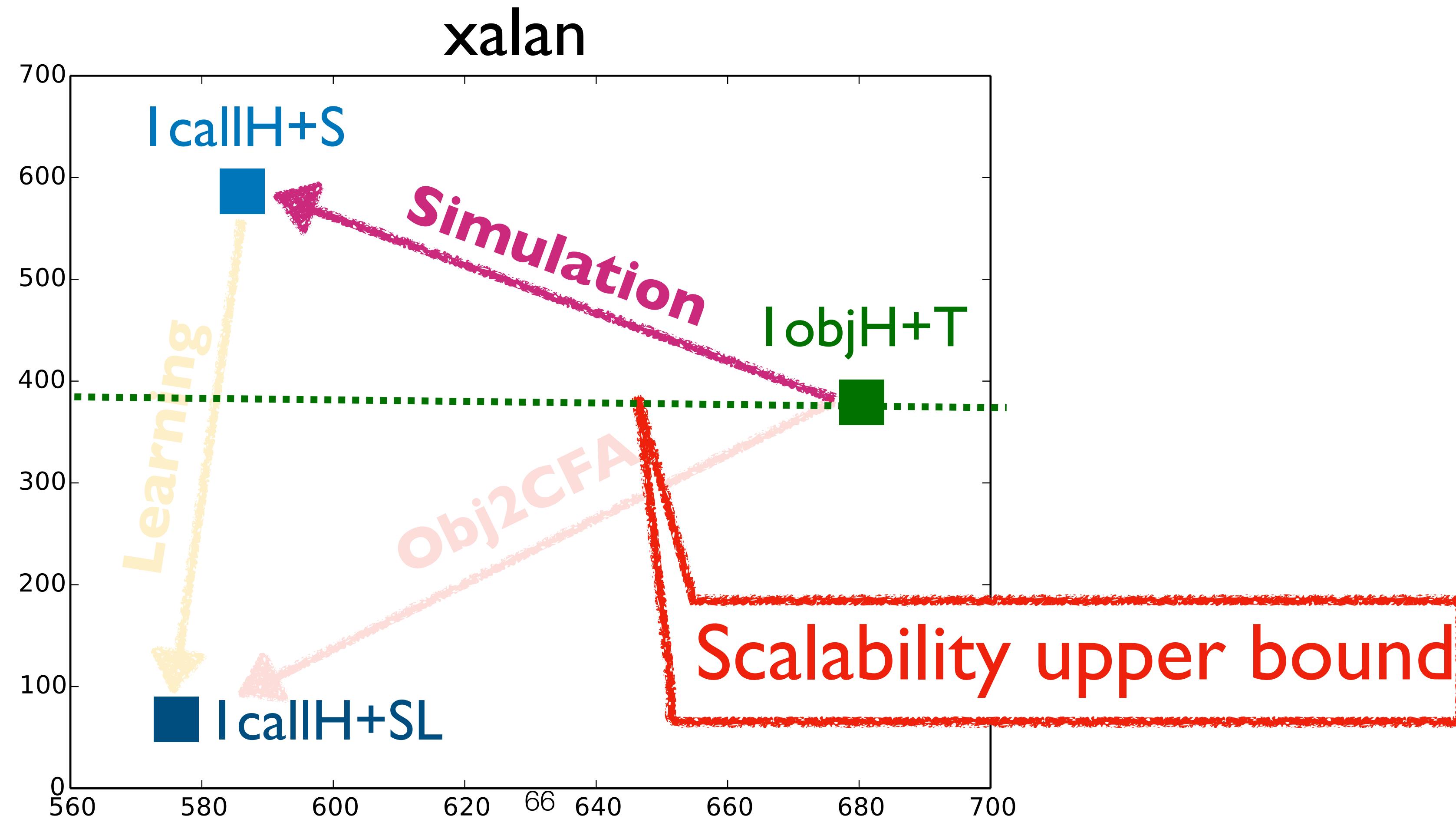
Our Technique : Obj2CFA

- **Obj2CFA** consists of **simulation** and simulation-guided **learning**



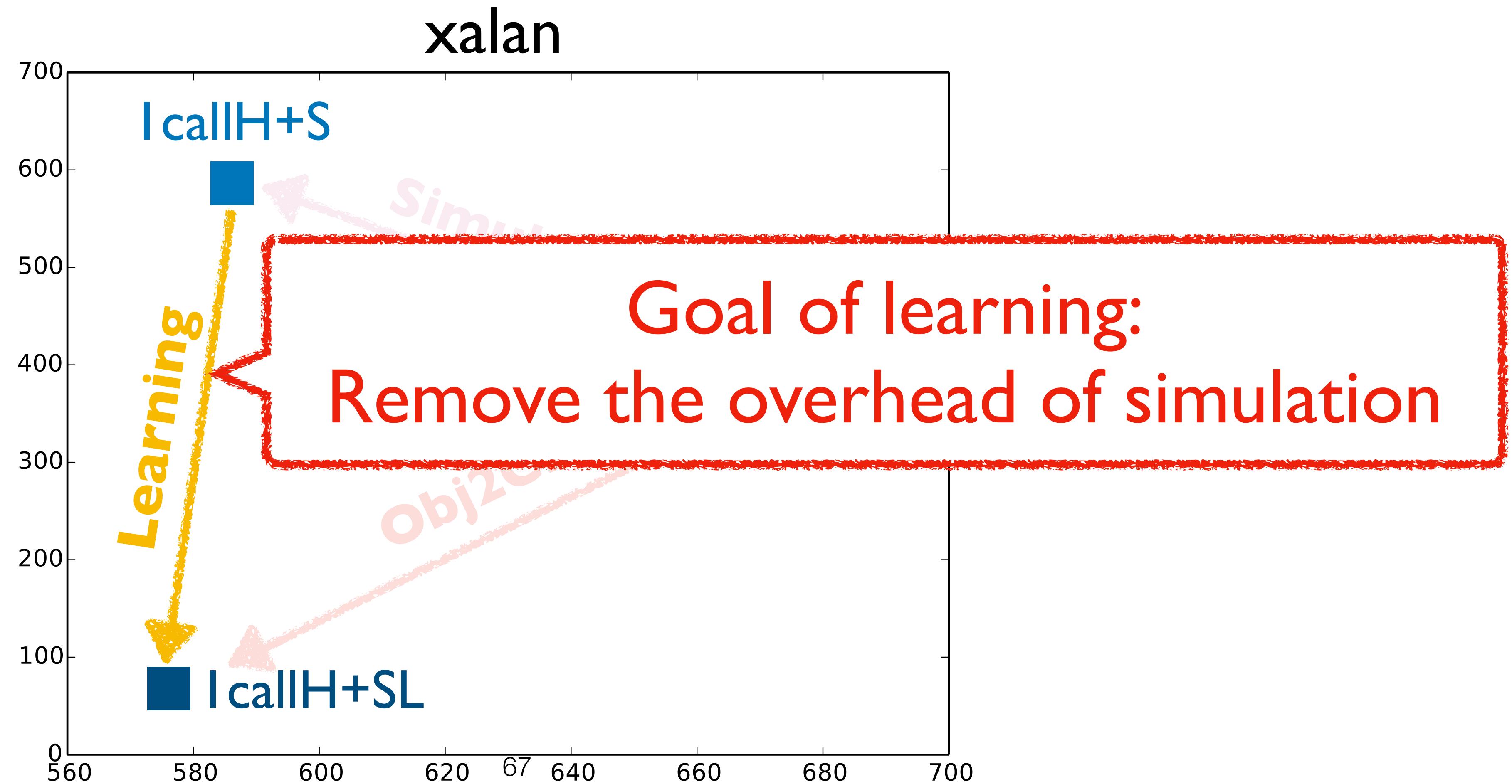
Our Technique : Obj2CFA

- **Obj2CFA** consists of **simulation** and simulation-guided **learning**



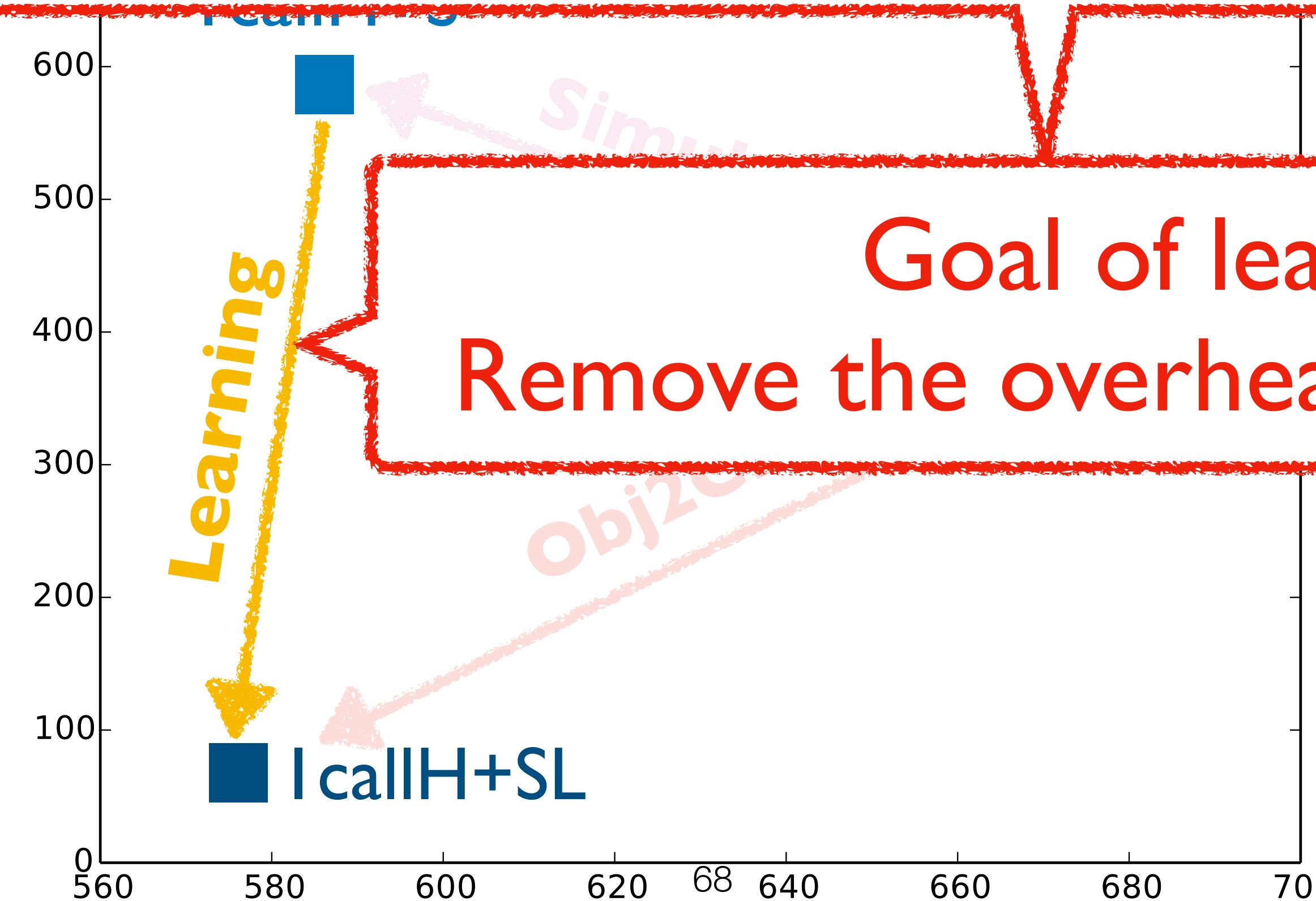
Our Technique : **Obj2CFA**

- **Obj2CFA** consists of **simulation** and simulation-guided **learning**



Our Technique: Obj2CEA

Given training programs and simulated tunneling abstractions, learning aims to find a model that produces similar tunneling abstractions without running the given object sensitivity



Our Technique · OHICFA

Given training programs and simulated tunneling abstractions,
learning aims to find a model that produces similar tunneling

The learned model will produce tunneling abstractions without
running object sensitivity

Details in paper

I call H+SL

Evaluation

Setting

- Doop
 - Pointer analysis framework for Java
 - Research Question: which one is better?

Call-site sensitivity vs Object sensitivity

Context tunneling is included

Setting

Doop

Negative results on CFA have been **repeatedly** reported on Doop

Strictly Declarative Specification of Sophisticated Points-to

**Pick Your Contexts Well: Understanding
The Making of a Precise and Scalable Pointe**

Yannis Smaragdakis
Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003, USA
and Department of Informatics,
University of Athens, 15784, Greece
yannis@cs.umass.edu—smaragd@di.uoa.gr

Martin Bravenboer
LogicBlox Inc.
Two Midtown Plaza
Atlanta, GA 30309, USA
martin.bravenboer@acm.org

Abstract

Object-sensitivity has emerged as an excellent context abstraction for points-to analysis in object-oriented languages. Despite its practical success, however, object-sensitivity is poorly understood. For instance, for a context depth of 2 or higher, past scalable implementations deviate significantly from the original definition of an object-sensitive analysis. The reason is that the analysis has many degrees of freedom, relating to which context elements are picked at every method call and object creation. We offer a clean model for the analysis design space, and discuss a formal and informal understanding of object-sensitivity and of how to create good object-sensitive analyses. The results are surprising in their extent: We find that past implementations have made a sub-optimal choice of contexts, to the severe detriment of precision and performance. We define a “full-object-sensitive” analysis that results in significantly higher precision, and often performance, for the exact same context depth. We also introduce “type-sensitivity” as an explicit approximation of object-sensitivity that preserves high context quality at substantially reduced cost. A type-sensitive points-to analysis makes an unconventional use of types as context: the context types are not dynamic types of objects involved in the analysis, but instead upper bounds on the dynamic types of their allocator objects. Our results expose the influence of context choice on the quality of points-to analysis and demonstrate type-sensitivity to be an idea with major impact: It decisively advances the state-of-the-art with a spectrum of analyses that simultaneously enjoy speed (several times faster than an analogous object-sensitive analysis), scalability (comparable to analyses with much less context-sensitivity), and precision (comparable to the best object-sensitive analysis with the same context depth).

Categories and Subject Descriptors F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis ; D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics

General Terms Algorithms, Languages, Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL ’11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11...\$10.00

1. Introduction

Points-to analysis (or *pointer analysis*) is one of the most fundamental static analysis problems. It consists of computing a static representation of the pointer expression (or just a variable) during program run-time. It is used by almost every other program analysis mechanism such as control flow analysis, type inference, and pointer determination in the presence of object-oriented dynamic structures such as closures and lambda applications. In fact, pointer analysis is the key to solving many other problems. Furthermore, the global nature of pointer analysis make it hard to interact with various functional languages, a feature that achieves tractable and efficient analysis. Pointer analysis consists of qualifying pointer expressions with information (e.g., “what is the value of *x*?”) over all possible states while separating all kinds of context-sensitive [18, 19] and *object-sensitivity*.

Ever since the introduction of pointer analysis [13], there has been significant research showing that it is a superior context abstraction yielding high precision. Object-sensitivity has been studied in various analyses have almost always been more precise than object-sensitive/KCFA-based analyses, and is concerned with understanding and maximizing its convenience and efficiency, making it even more scalable and efficient.

What is object-sensitivity? What is the best way to describe the context abstraction? What is the better-known call-site sensitivity? What is the method (the method) as context abstraction? How does it separate information between objects? How does it per-call-stack (i.e., sequential) context abstraction? What is the effect that led to the current result? What is the effect of the information on heap objects? What is the effect that led to the object’s behavior? What is the effect below, a 1-call-site sensitivity?

Hybrid Context-Sensitivity for Points-To Analysis

George Kastrinis Yannis Smaragdakis

Department of Informatics

University of Athens

{gkastrinis,smaragd}@di.uoa.gr

Abstract

Context-sensitive points-to analysis is valuable for achieving high precision with good performance. The standard flavors of context-sensitivity are call-site-sensitivity (kCFA) and object-sensitivity. Combining both flavors of context-sensitivity increases precision but at an infeasibly high cost. We show that a selective combination of call-site- and object-sensitivity for Java points-to analysis is highly profitable. Namely, by keeping a combined context only when analyzing selected language features, we can closely approximate the precision of an analysis that keeps both contexts at all times. In terms of speed, the selective combination of both kinds of context not only vastly outperforms non-selective combinations but is also faster than a mere object-sensitive analysis. This result holds for a large array of analyses (e.g., 1-object-sensitive, 2-object-sensitive with a context-sensitive heap, type-sensitive) establishing a new set of performance/precision sweet spots.

Categories and Subject Descriptors F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis; D.3.4 [Programming Languages]: Processors—Compilers

General Terms Algorithms, Languages, Performance

Keywords points-to analysis; context-sensitivity; object-sensitivity; type-sensitivity

1. Introduction

Points-to analysis is a static program analysis that consists of computing all objects (typically identified by allocation site) that a program variable may point to. The area of points-to analysis (and its close relative, *alias analysis*) has been the focus of intense research and is among the most standardized and well-understood of inter-procedural analyses. The emphasis of points-to analysis algorithms is on combining fairly precise modeling of pointer behavior with scalability. The challenge is to pick judicious approximations that will allow satisfactory precision at a reasonable cost. Furthermore, although increasing precision often leads to higher asymptotic complexity, this worst-case behavior is rarely encountered in actual practice. Instead, techniques that are effective at maintaining good precision often also exhibit better average-case performance, since smaller points-to sets lead to less work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

Copyright © 2013 ACM 978-1-4503-2014-6/13/06...\$15.00

One of the major tools for exploitation/performance tradeoff has been sensitivity consists of qualifying local context information: the analysis uses same context value, while separating different contexts. This approach tries to naturally results in any static analysis from different dynamic program paths sensitivity have been explored in the [22, 23] and *object-sensitivity* [18, 19].

A call-site-sensitive/kCFA analyzes labels of instructions that may call the method. That is, the analysis separates information about method arguments (per call-stack (i.e., method invocations that led to the current analysis) separates information about method invocations that led to the object in the code example below, a 1-call-to *foo* on lines 7 and 9. This means *foo* separately for two cases: that if it points to anything *obj1* may point to, and *obj2* may point to.

```
1 class C {
2     void foo(Objet o) { ... }
3 }
4
5 class Client {
6     void bar(C c1, C c2) {
7         ...
8         c1.foo(obj1);
9         ...
10        c2.foo(obj2);
11    }
}
```

In contrast, object-sensitivity uses object of instructions containing a new state. (Hence, a better name for “object-allocation-site sensitivity”.) That is, an object, the analysis separates the allocation site of the receiver object (the method is called), as well as the context. Thus, in the above example will analyze *foo* separately depending on whether *c1* and *c2* may point to. It does not matter whether *c1* and *c2* are local to how many objects: the allocation may be remote and unrelated to the analysis. It is not possible to compare the precision of a call-site-sensitive analysis in practice, and it is not even clear whether the object set of all calls to *foo* as one case, as two,

Introspective Analysis: Context-Sensitivity, Across the Board

Yannis Smaragdakis George Kastrinis George Balatsouras

Department of Informatics
University of Athens
{smaragd,gkastrinis,gbalats}@di.uoa.gr

Abstract

Context-sensitivity is the primary approach for adding more precision to a points-to analysis, while hopefully also maintaining scalability. An oft-reported problem with context-sensitive analyses, however, is that they are bi-modal: either the analysis is precise enough that it manipulates only manageable sets of data, and thus scales impressively well, or the analysis gets quickly derailed at the first sign of imprecision and becomes orders-of-magnitude more expensive than would be expected given the program's size. There is currently no approach that makes precise context-sensitive analyses (of any flavor: call-site-, object-, or type-sensitive) scale across the board at a level comparable to that of a context-insensitive analysis. To address this issue, we propose introspective analysis: a technique for uniformly scaling context-sensitive analysis by eliminating its performance-debilitating behavior, at a small precision expense. Introspective analysis consists of a common adaptivity pattern: first perform a context-insensitive analysis, then use the results to selectively refine (i.e., analyze context-sensitively) program elements that will not cause explosion in the running time space. The technical challenge is to appropriately identify such program elements. We show that a simple but principled approach can be remarkably effective, achieving scalability (often with dramatic speedup) for benchmarks previously completely out-of-reach for deep context-sensitive analyses.

Categories and Subject Descriptors F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis; D.3.4 [Programming Languages]: Processors—Compilers

General Terms Algorithms, Languages, Performance

Keywords points-to analysis; context-sensitivity; object-sensitivity; type-sensitivity

Introduction

Points-to analysis is probably the most common whole-program static analysis, and often serves as a substrate for a variety of high-level program analysis tasks. Points-to analysis computes the set of objects (abstracted as their allocation sites) that a program variable may point to during runtime. The promise, as well as the challenge,

is to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or to publish, to post on servers or to redistribute to lists, requires prior specific permission or a fee. Request permissions from permissions@acm.org.

DI'14, June 9–11, 2014, Edinburgh, United Kingdom.
Copyright © held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-2784-8/14/06...\$15.00.
dx.doi.org/10.1145/2594291.2594320

of points-to analysis is to yield usefully precise results while sacrificing scalability: the analysis inputs are algorithms are typically quadratic or cubic, near-linear behavior in practice, by exploiting and maintaining precision. Indeed precision can go hand-in-hand in a good points-to analysis: smaller points-to sets lead to less work.

Context-sensitivity is a common way of achieving scalability in points-to analysis. It consists of abstractions and objects with context information: formation (e.g., “what objects this method analyzes over all possible executions that map to them?”), while separating executions that map to the same objects. Context-sensitivity attempts to avoid precisely the behavior of different dynamic programs. Context-sensitivity comes in many flavors, depending on the information, such as *call-site-sensitivity* [22, 19, 20], and *type-sensitivity* [24].

An oft-remarked fact about context-sensitive analyses is that even the best algorithms have a common failing: they cannot maintain precision. Past literature remarks of a [...] deep-context analysis is brittle: sensitive analyses have been associated with contexts” [15]; “algorithms completely hit a precision ceiling, with the number of tuples exploding exponentially. Recent published results [12] fail to run a 2-solver in under 90mins for 2 of 10 DaCapo benchmarks, which take more than 1,000secs, although benchmarks of similar or larger size get analyzed in seconds.

Thus, when context-sensitivity works, it is terms of both precision and performance. When it fails miserably, quickly exploding in cost. Context-insensitive analyses uniformly scale well. Figure 1 vividly demonstrates this phenomenon: the DaCapo benchmarks, analyzed with the Doop context-insensitive (insens) analysis and a 2-solver with a context-sensitive heap (2objIH). The analysis time of the longest-running benchmarks (qlquad and jython, timed out after 90mins and would not terminate even for much longer) can be seen, context-insensitive analyses vary in performance, while context-sensitivity often causes memory usage to explode.

Faced with this unpredictability of context-sensitivity, the common reaction is to avoid it, favoring conservative, well-behaved programs. Even worse, for chewing expensive context-sensitivity is not always the best choice: it is often less precise than context-insensitive analysis is just not good enough [4] and academic researchers [3] alike.

Making k -Object-Sensitive Pointer Analyses More Precise with Still k -Limiting

Tian Tan¹, Yue Li¹, and Jingling Xue^{1,2}

¹ School of Computer Science and Engineering, UNSW Australia
² Advanced Innovation Center for Imaging Technology, CNU, China

Abstract. Object-sensitivity is regarded as arguably the best context abstraction for pointer analysis in object-oriented languages. However, a k -object-sensitive pointer analysis, which uses a sequence of k allocation sites (as k context elements) to represent a calling context of a method call, may end up using some context elements redundantly without inducing a finer partition of the space of (concrete) calling contexts for the method call. In this paper, we introduce BEAN, a general approach for improving the precision of any k -object-sensitive analysis, denoted $k\text{-}obj$, by still using a k -limiting context abstraction. The novelty is to identify allocation sites that are redundant context elements in $k\text{-}obj$ from an Object Allocation Graph (OAG), which is built based on a pre-analysis (e.g., a context-insensitive Andersen's analysis) performed initially on a program and then avoid them in the subsequent k -object-sensitive analysis for the program. BEAN is generally more precise than $k\text{-}obj$, with a precision that is guaranteed to be as good as $k\text{-}obj$ in the worst case. We have implemented BEAN as an open-source tool and applied it to refine two state-of-the-art whole-program pointer analyses in DOOP. For two representative clients (*may-alias* and *may-fail-cast*) evaluated on a set of nine large Java programs from the DaCapo benchmark suite, BEAN has succeeded in making both analyses more precise for all these benchmarks under each client at only small increases in analysis cost.

1 Introduction

Pointer analysis, as an enabling technology, plays a key role in a wide range of client applications, including bug detection [3, 25, 35, 34], security analysis, compiler optimisation [6, 33], and program understanding [12]. Two main dimensions of pointer analysis precision are flow-sensitivity and context-sensitivity. For C/C++ programs, flow-sensitivity is needed by many clients [11, 16, 31]. For object-oriented programs, e.g., Java programs, however, context-sensitivity is known to deliver trackable and useful precision [17, 19–21, 28–30], in general.

There are two general approaches to achieving context-sensitivity for object-oriented programs, call-site-sensitivity (k -CFA) [27] and object-sensitivity [24, 29] (among others). A k -CFA analysis represents a calling context of a method call by using a sequence of k call sites (i.e., k labels with each denoting a statement). In contrast, a k -object-sensitive analysis uses k object allocation sites (k labels with each denoting a new statement) as context elements.

We present our work for Java, other programming languages, most of which are design-oriented, overcomes the context heuristics, and context-awareness. We improve the analysis experiments, and CCS Conference on Chinese Information Processing. Additionally, ACM SIGART, Sehun, and Analytical methods for the study of subprograms. Finally, the first author provides the full abstract, prior studies, and © 2017. The first author retains the copyright to this article.

2475-123X/17/0300-0001-12 \$15.00 © 2017 IEEE. <https://doi.org/10.1109/ICCCN.2017.8210301>

JUN JEONG, Korea University, Republic of Korea
SEOK JEON^{*}, Korea University, Republic of Korea
GDEOK CHA, Korea University, Republic of Korea
HAKJOO OH[†], Korea University, Republic of Korea

present a new data-driven approach to achieve highly cost-effective context-sensitive points-to analysis. While context-sensitivity has greater impact on the analysis precision and performance than any precision-improving techniques, it is difficult to accurately identify the methods that would benefit the most from context-sensitivity and decide how much context-sensitivity should be used for them. Manually defining such rules is a nontrivial and laborious task that often delivers suboptimal results in practice. To overcome these challenges, we propose an automated and data-driven approach that learns to effectively apply context-sensitivity from codebases. In our approach, points-to analysis is equipped with a parameterized set of heuristic rules, in disjunctive form of properties on program elements, that decide when and how much to apply context-sensitivity. We present a greedy algorithm that efficiently learns the parameter of the heuristic rules. We implemented our approach in the Doop framework and evaluated using three types of context-sensitive analyses: conventional object-sensitivity, selective hybrid object-sensitivity, and type-sensitivity. In all cases, our experimental results show that our approach significantly outperforms existing techniques.

Concepts: • Theory of computation → Program analysis; • Computing methodologies → Machine learning approaches;

Keywords: Data-driven program analysis, Points-to analysis, Context-sensitivity

Reference Format:
Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-Driven Context-Sensitivity for Points-to Analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 100 (October 2017), 27 pages.
[//doi.org/10.1145/3133924](https://doi.org/10.1145/3133924)

INTRODUCTION

Points-to analysis is one of the most important static program analyses. It approximates various memory locations that a pointer variable may point to at runtime. While useful as a stand-alone tool for many program verification tasks (e.g., detecting null-pointer dereferences), it is a key ingredient for subsequent higher-level program analyses such as static bug-finders, security auditing tools, and program understanding tools.

In object-oriented languages, context-sensitive points-to analysis is important as it must distinguish a method's local variables and objects in different calling-contexts. For languages like Java, the first and second authors contributed equally to this work and the corresponding author.

^{*}Authors' email addresses: S. Jeong, gifaranga@korea.ac.kr; M. Jeon, minseok_jeon@korea.ac.kr; S. Cha, scha@korea.ac.kr; H. Oh, hakjoo_oh@korea.ac.kr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee and without permission from the copyright holders for persons holding a valid ACM Digital Library license. Copying for general distribution or for commercial advantage is permitted only with the written permission of the copyright holders. Requests for further permission or information should be addressed to permissions@acm.org.

© Association for Computing Machinery.
421/2017/10-ART100
[//doi.org/10.1145/3133924](https://doi.org/10.1145/3133924)

2009
(OOPSL

2011 (POPL)

2013 (PLDI)

2014 (PLDI)

2016 (SAS)

2017 OOPSLA)

Setting

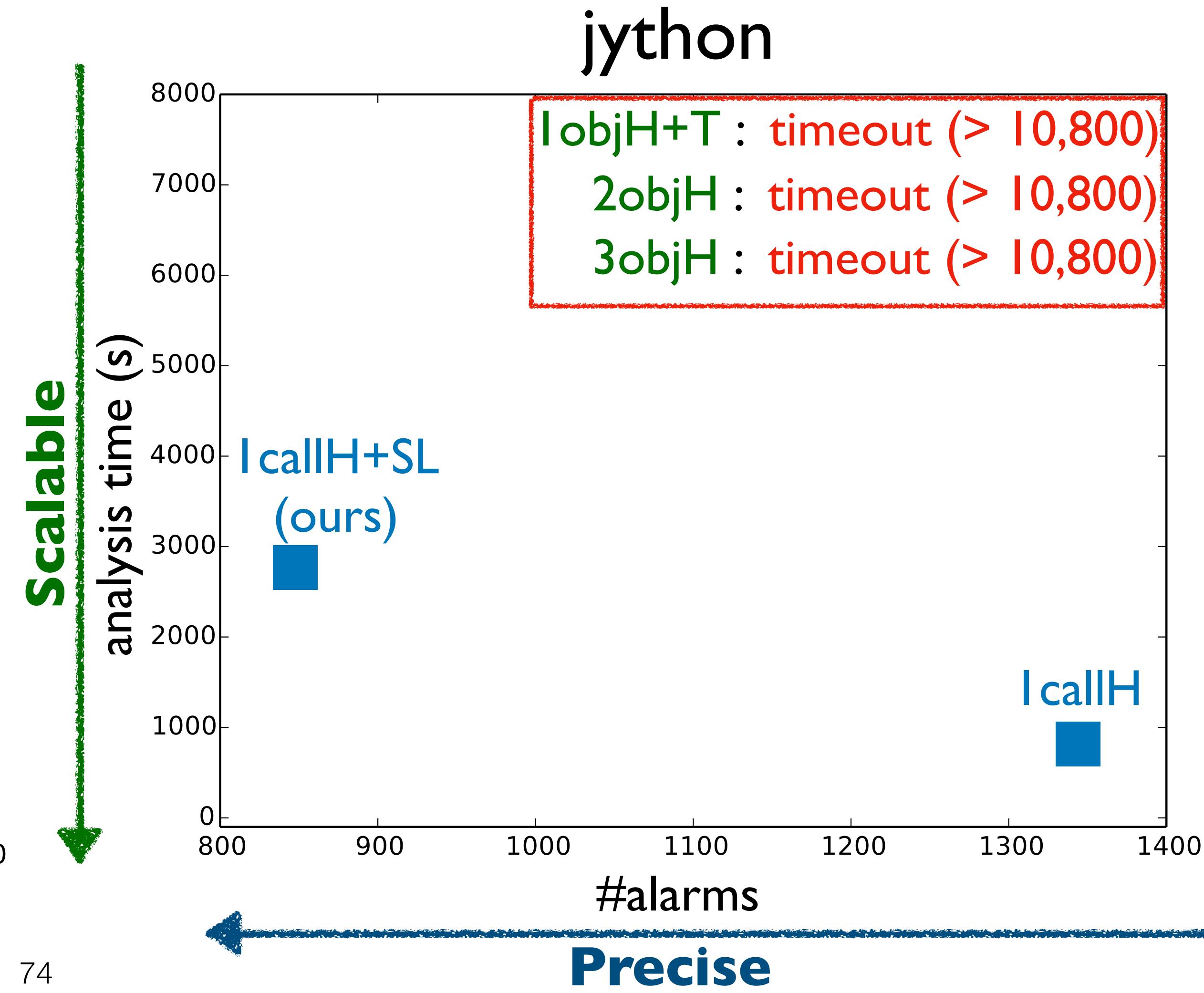
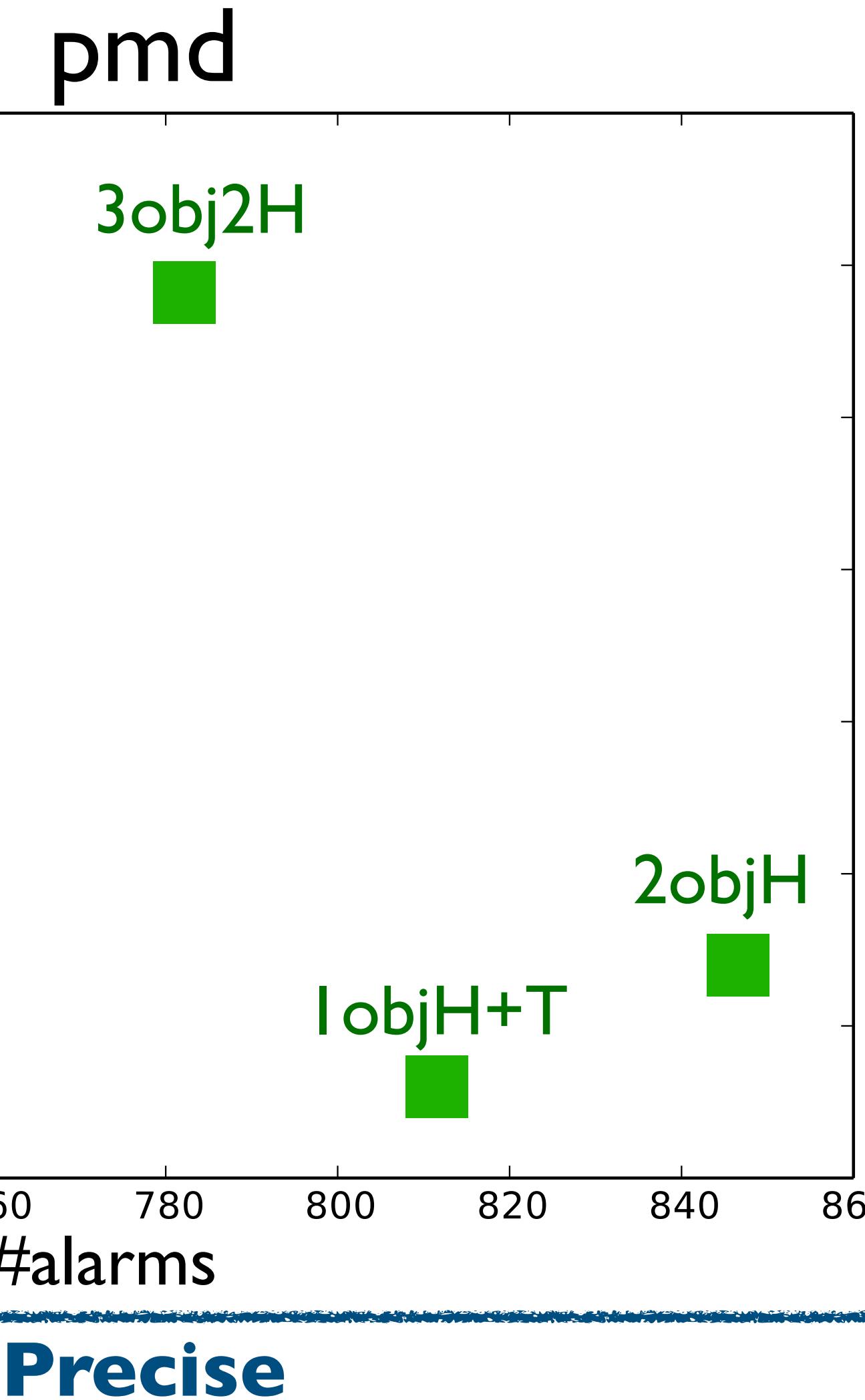
- Doop
 - Pointer analysis framework for Java
 - Research Question: which one is better?

Call-site sensitivity vs Object sensitivity

Context tunneling is included

Call-site Sensitivity vs Object Sensitivity

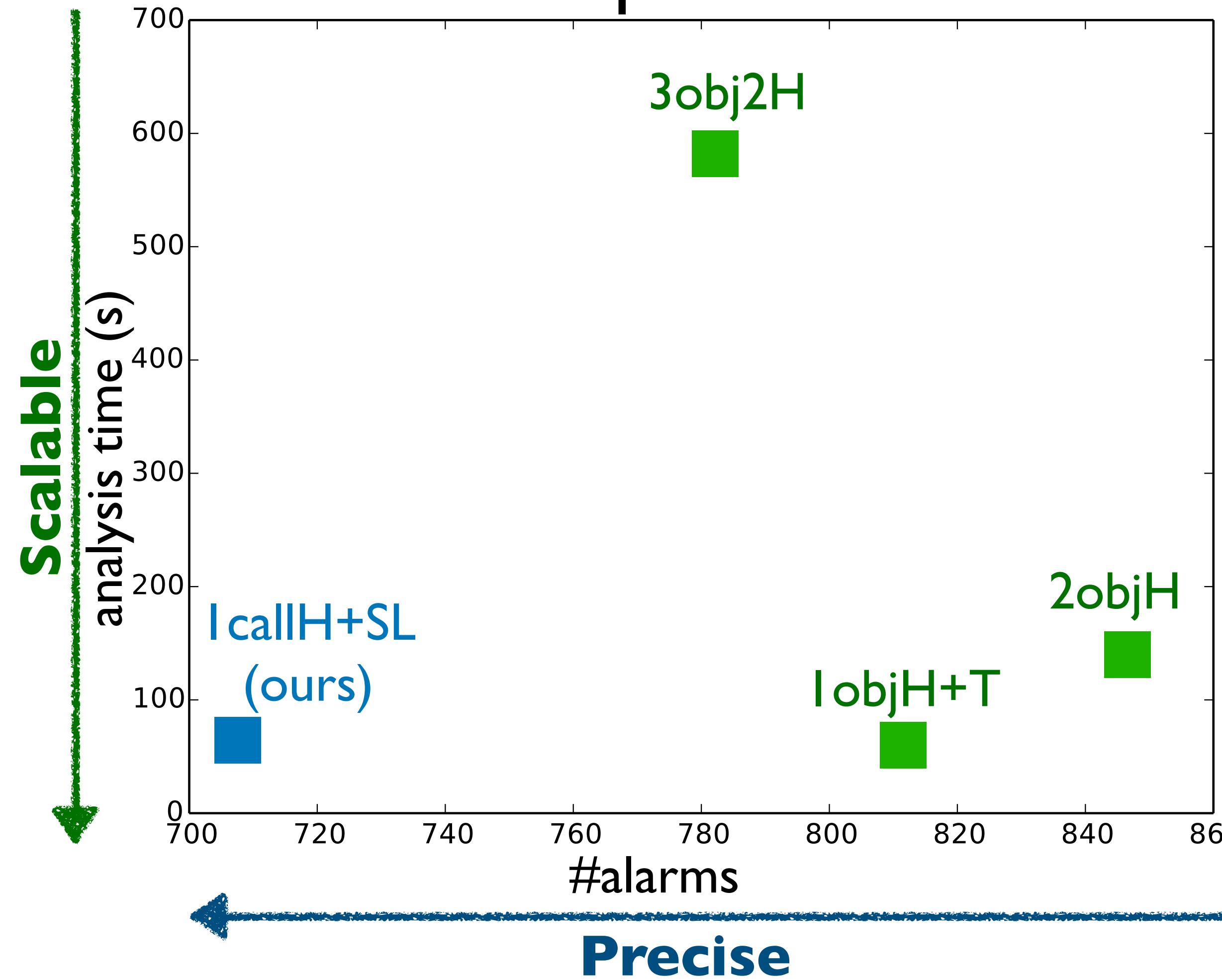
- $I_{callH+SL}$ (ours) is more precise and scalable than the existing object sensitivities



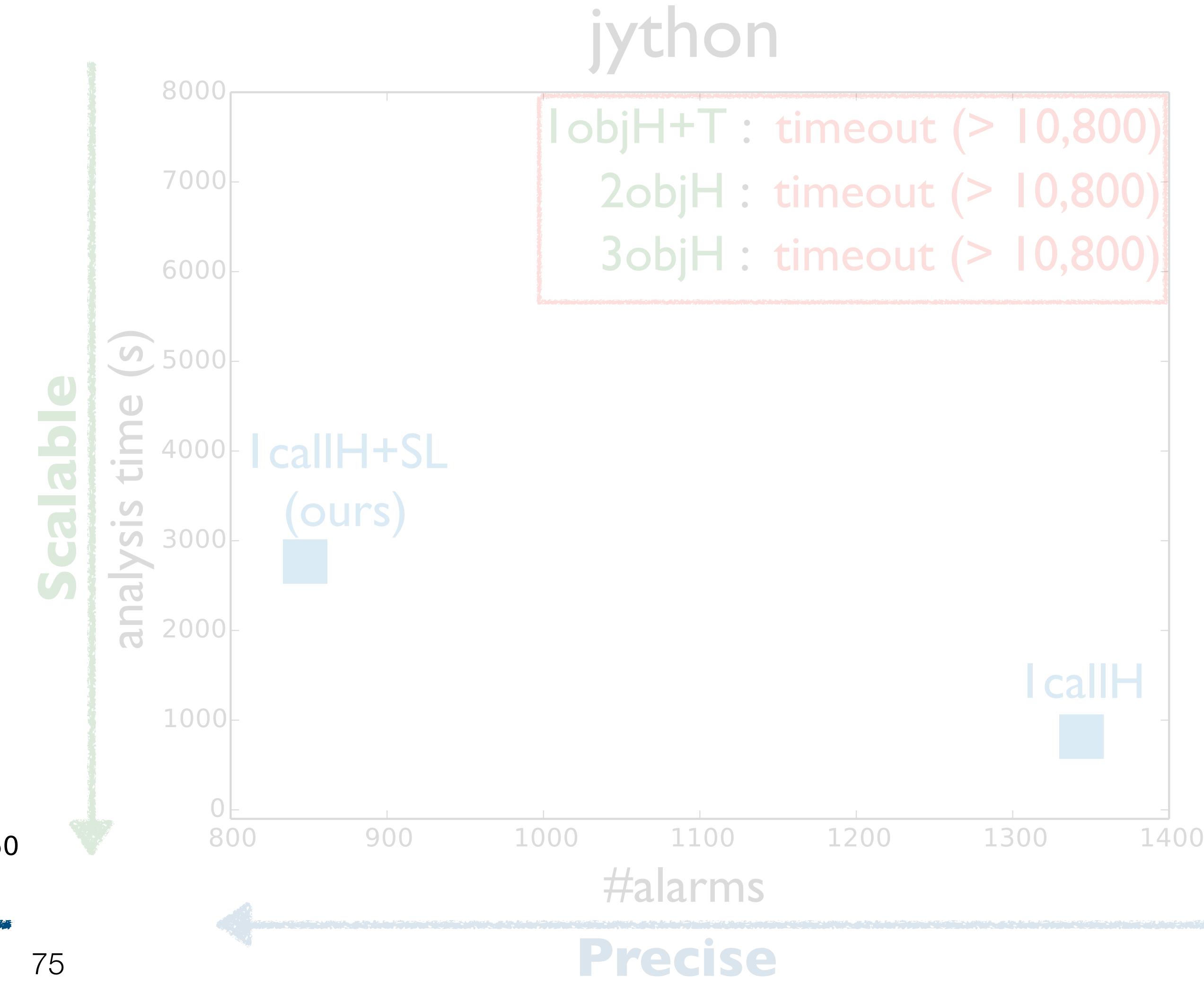
Call-site Sensitivity vs Object Sensitivity

- $I_{callH+SL}$ (ours) is **more precise** and **scalable** than the existing object sensitivities

pmd

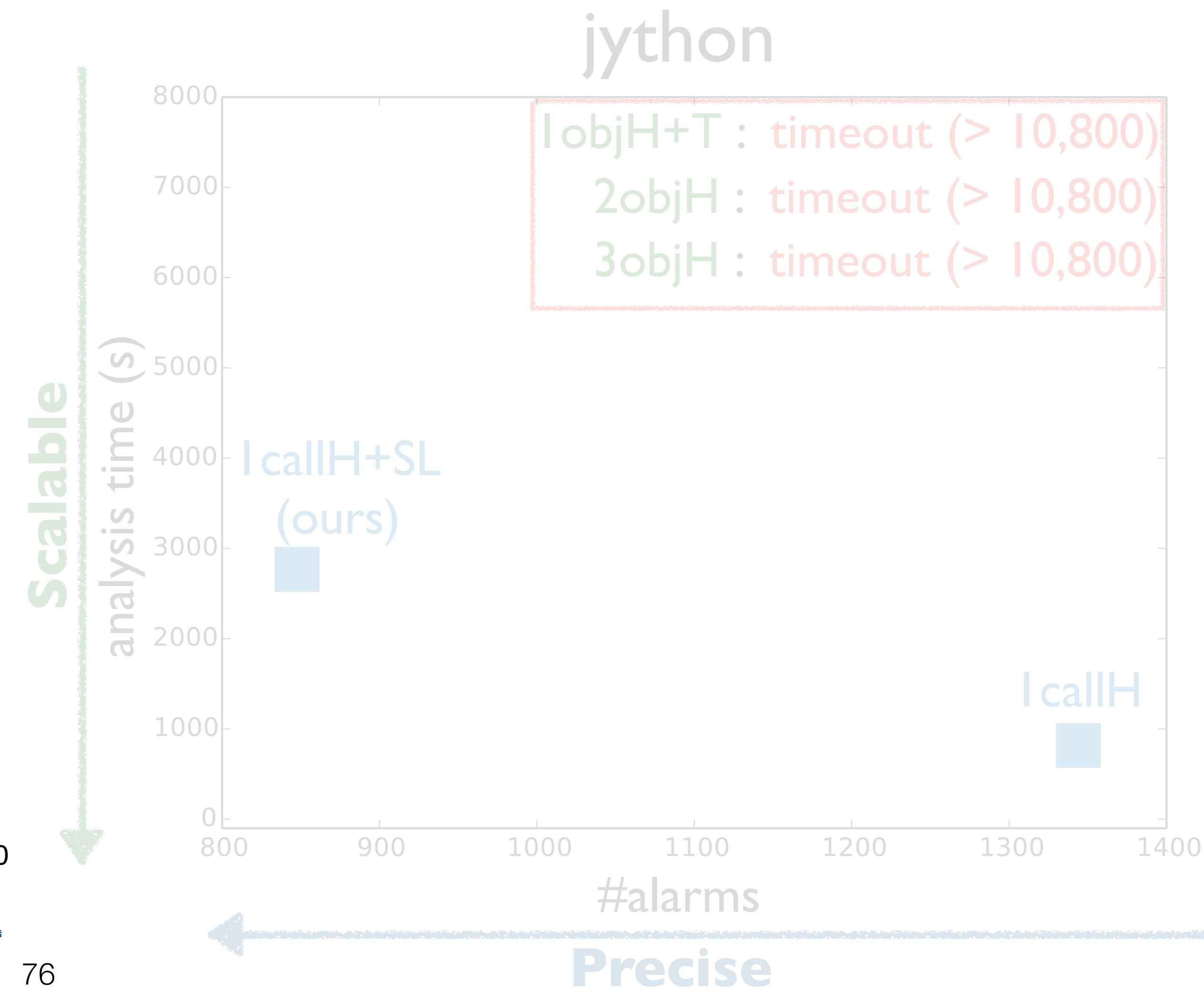
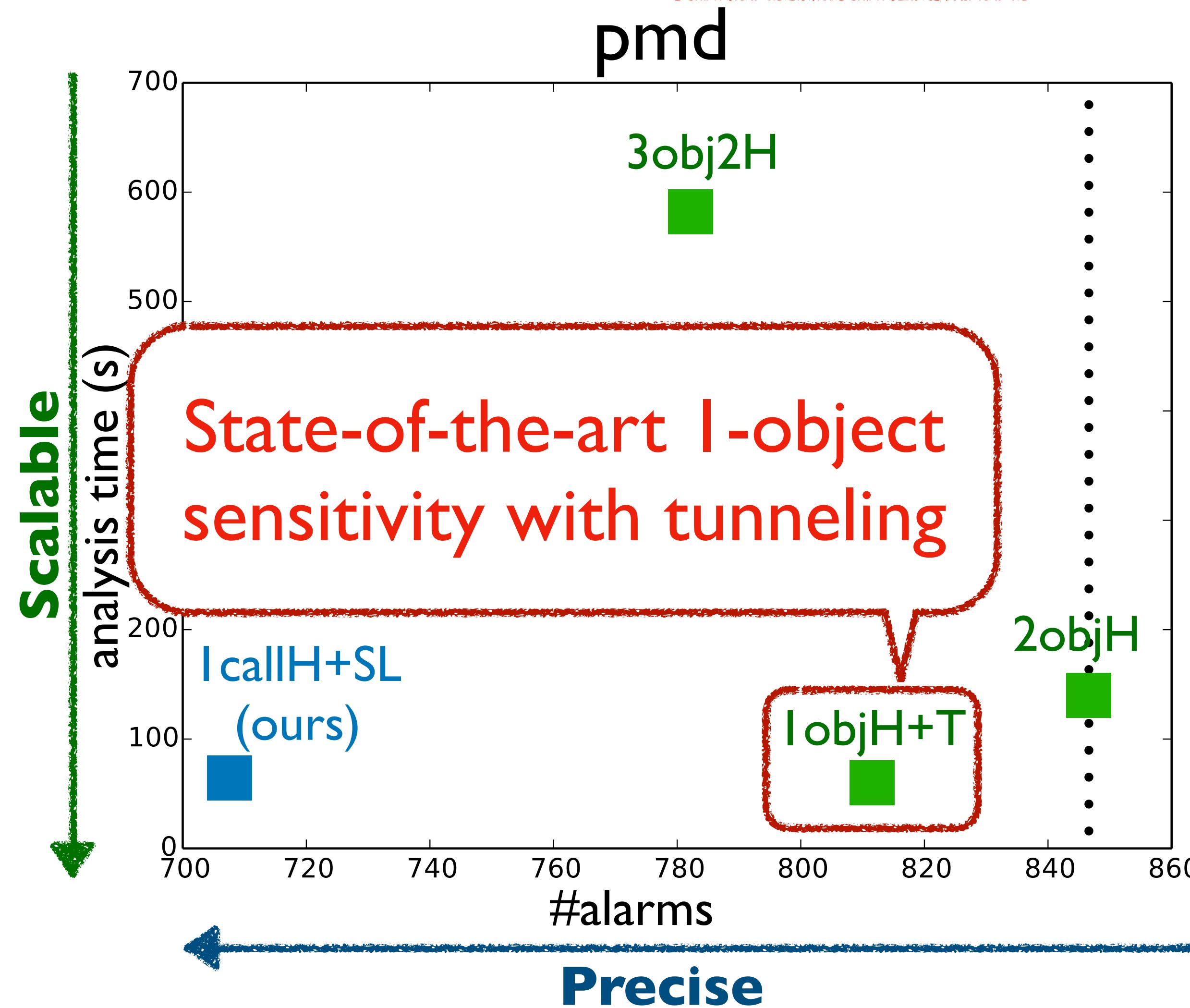


jython



Call-site Sensitivity vs Object Sensitivity

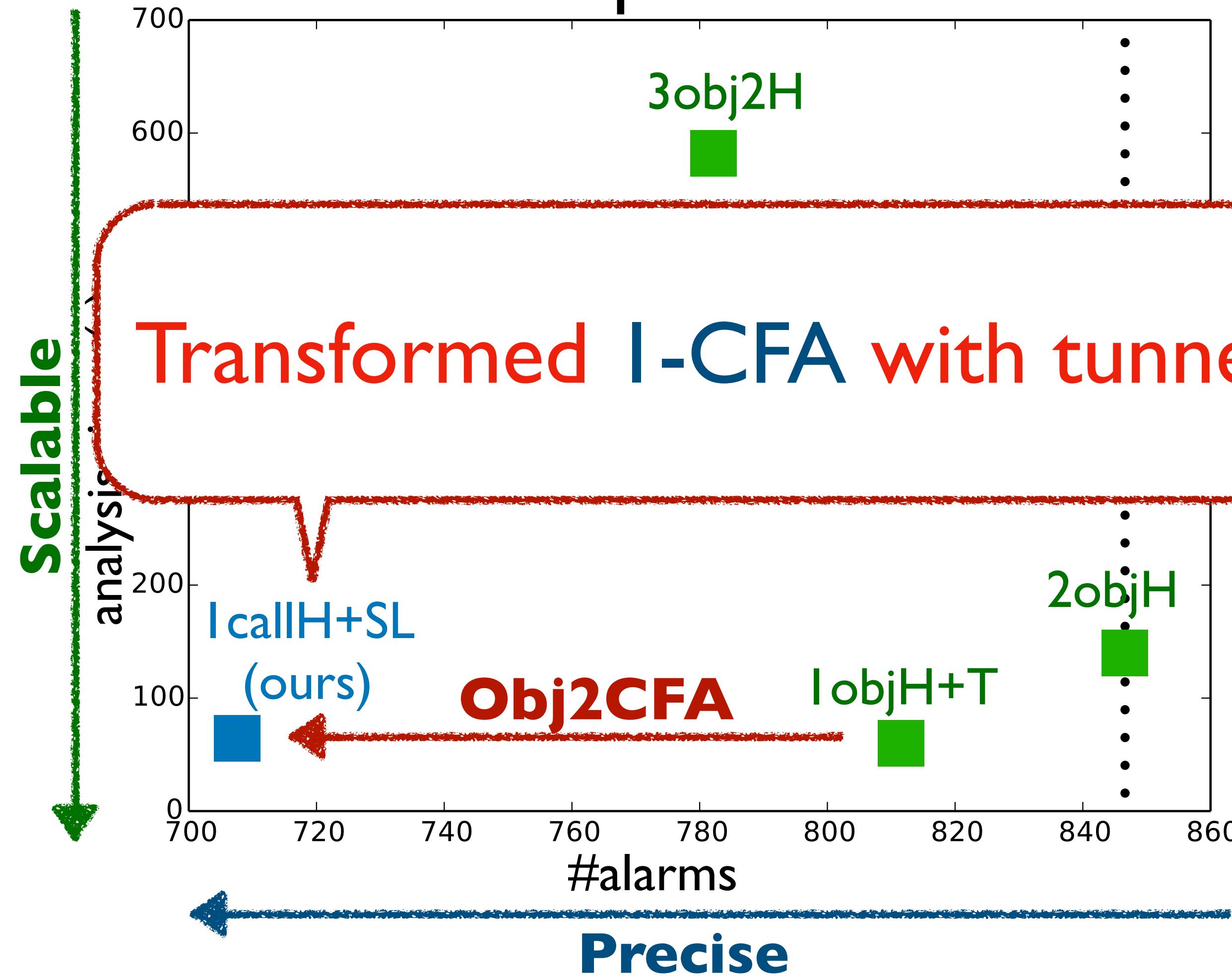
- $I_{callH+SL}$ (ours) is **more precise** and **scalable** than the existing object sensitivities



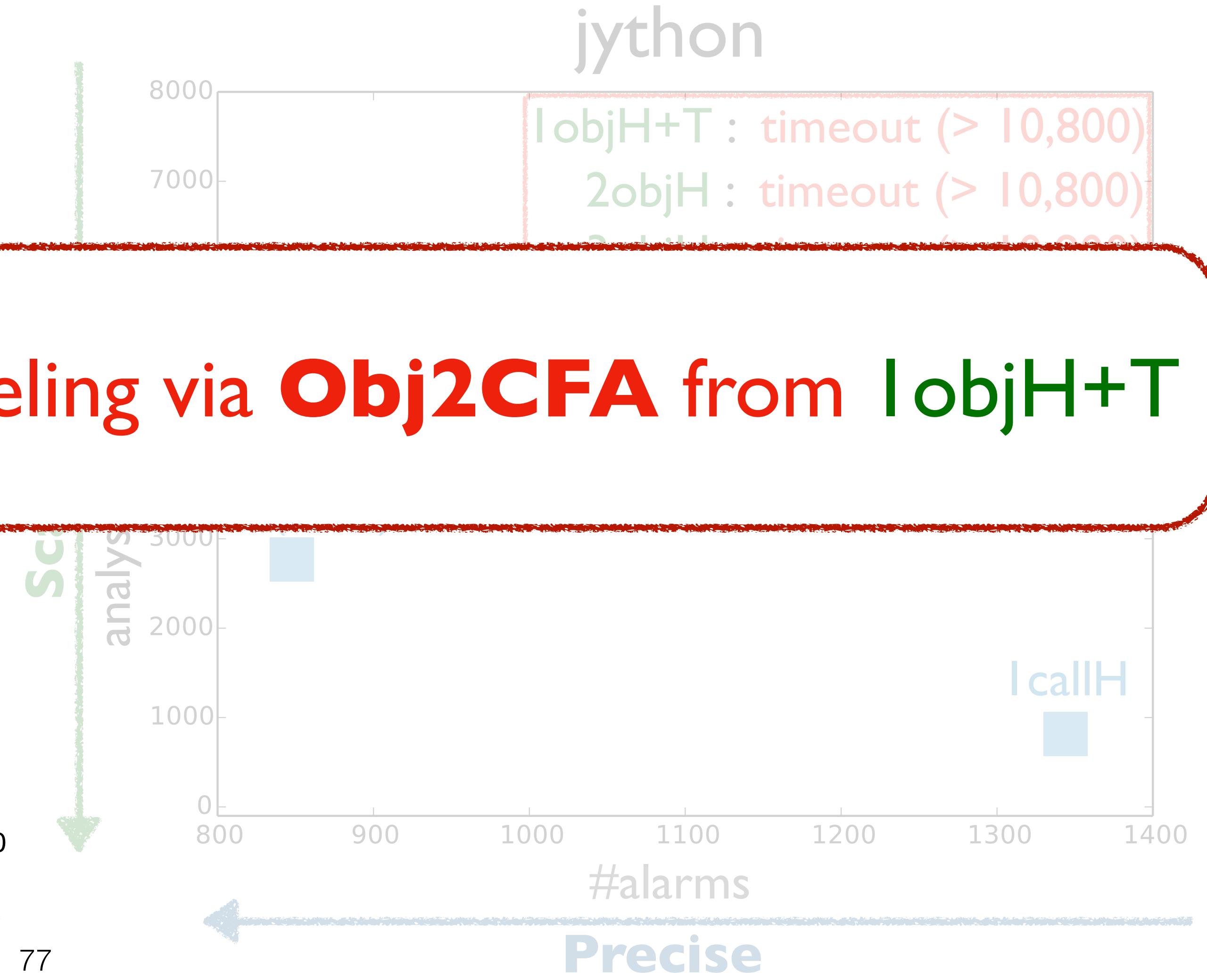
Call-site Sensitivity vs Object Sensitivity

- IcallH+SL (ours) is **more precise** and **scalable** than the existing object sensitivities

pmd

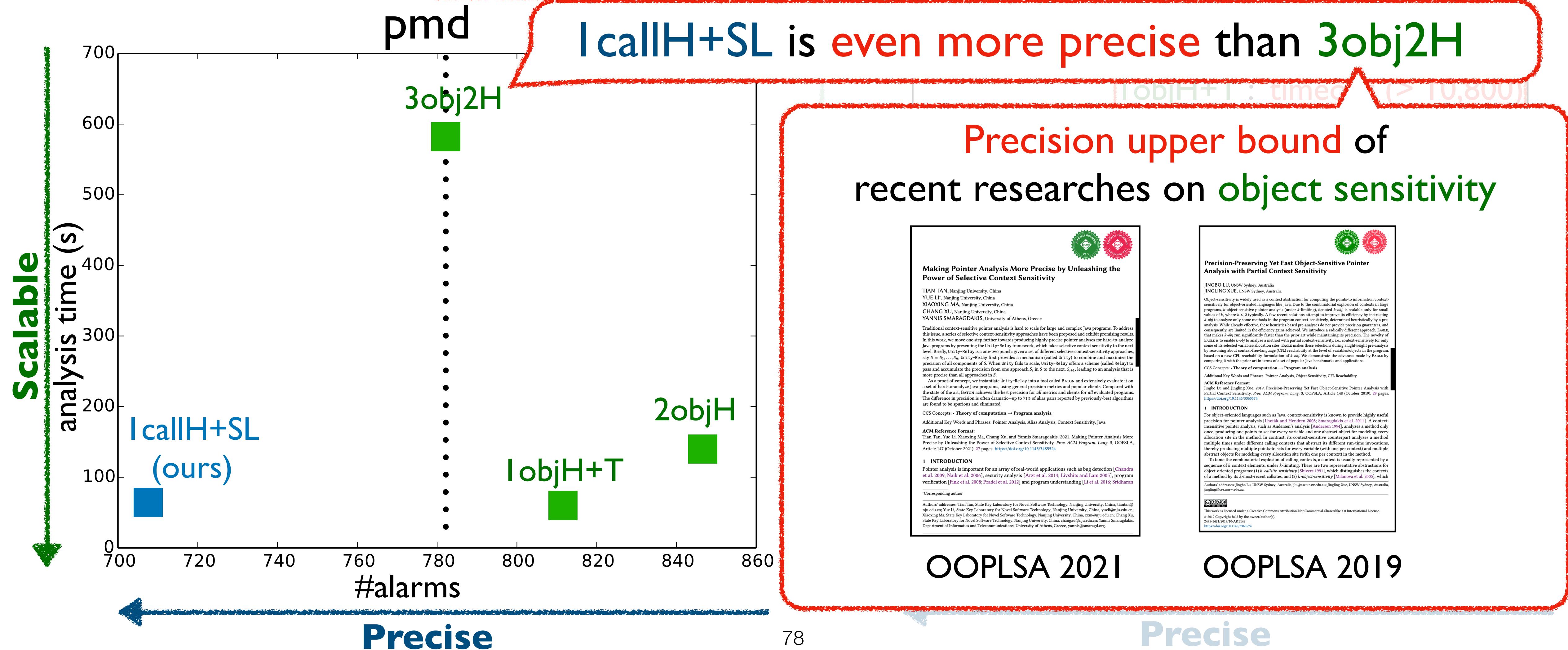


jython



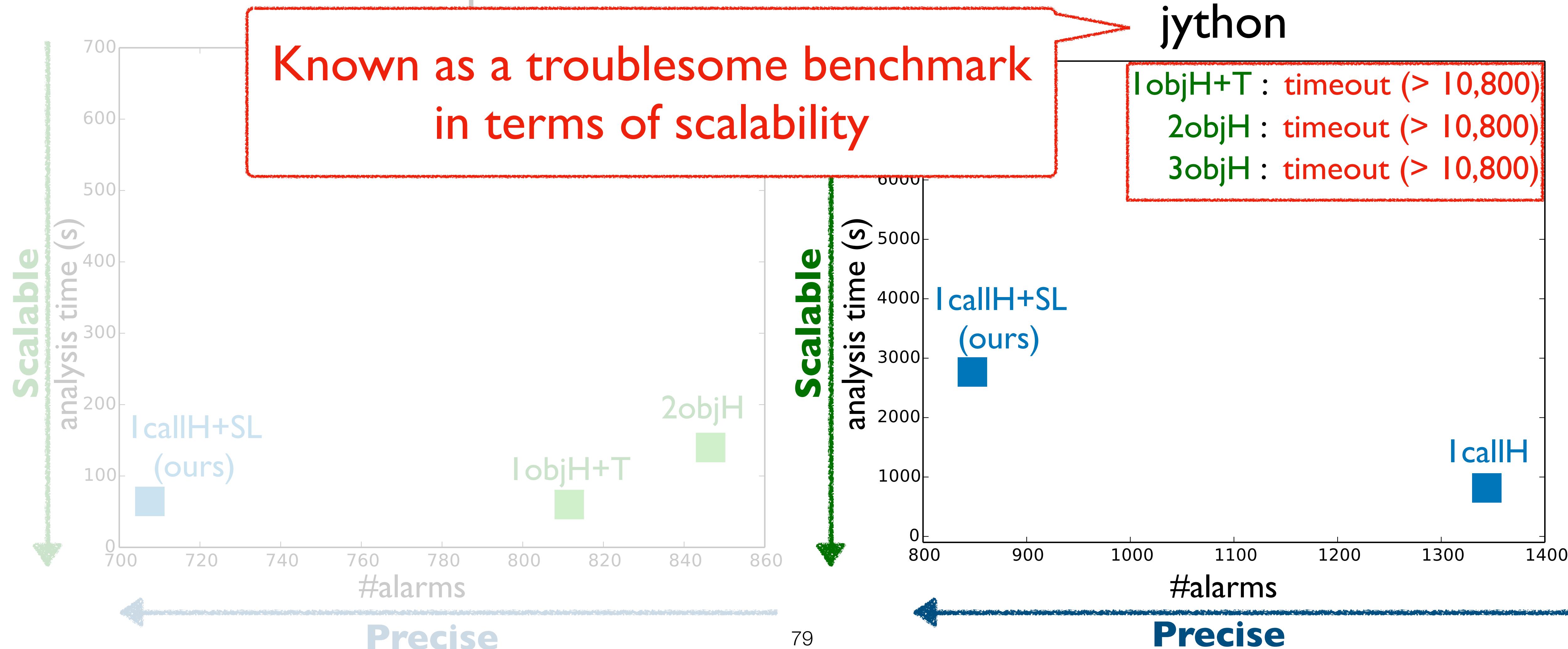
Call-site Sensitivity vs Object Sensitivity

- IcallH+SL (ours) is more precise and scalable than the existing object sensitivities



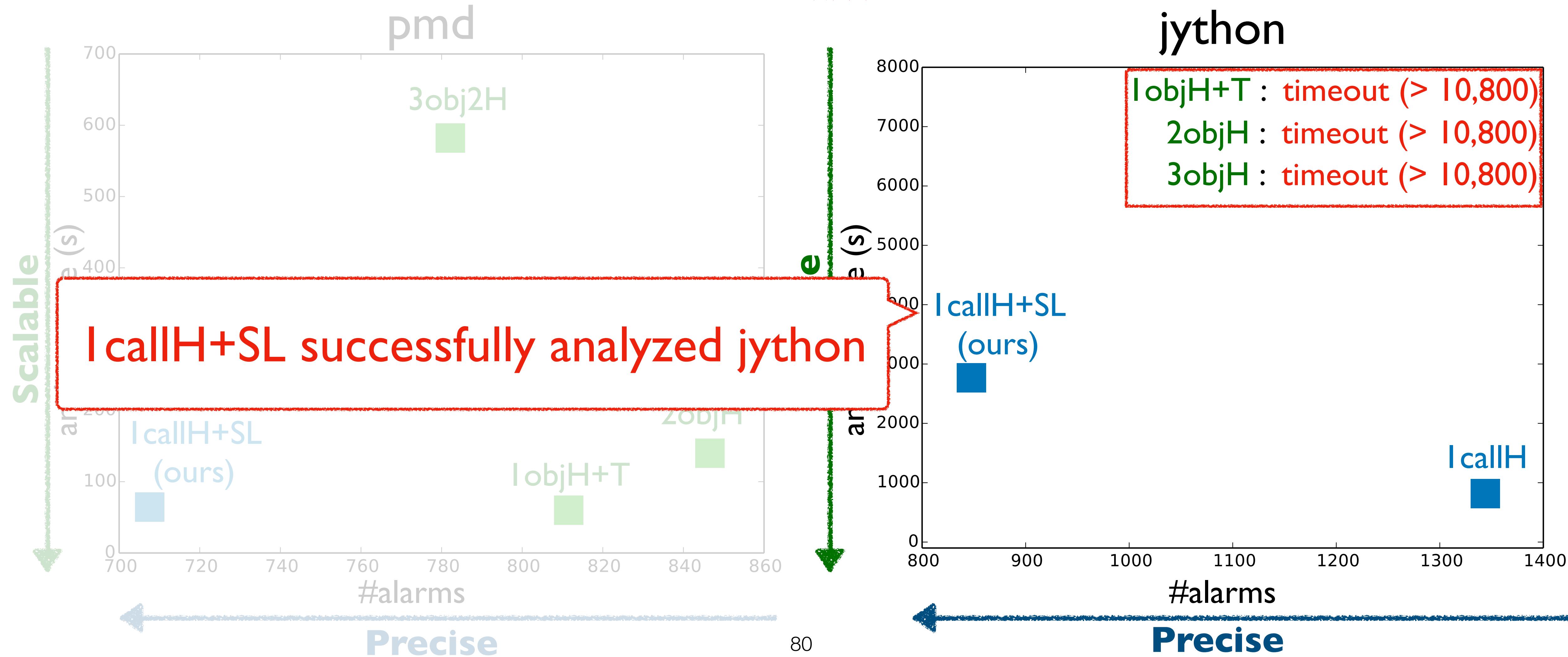
Call-site Sensitivity vs Object Sensitivity

- $I_{callH+SL}$ (ours) is more precise and scalable than the existing object sensitivities



Call-site Sensitivity vs Object Sensitivity

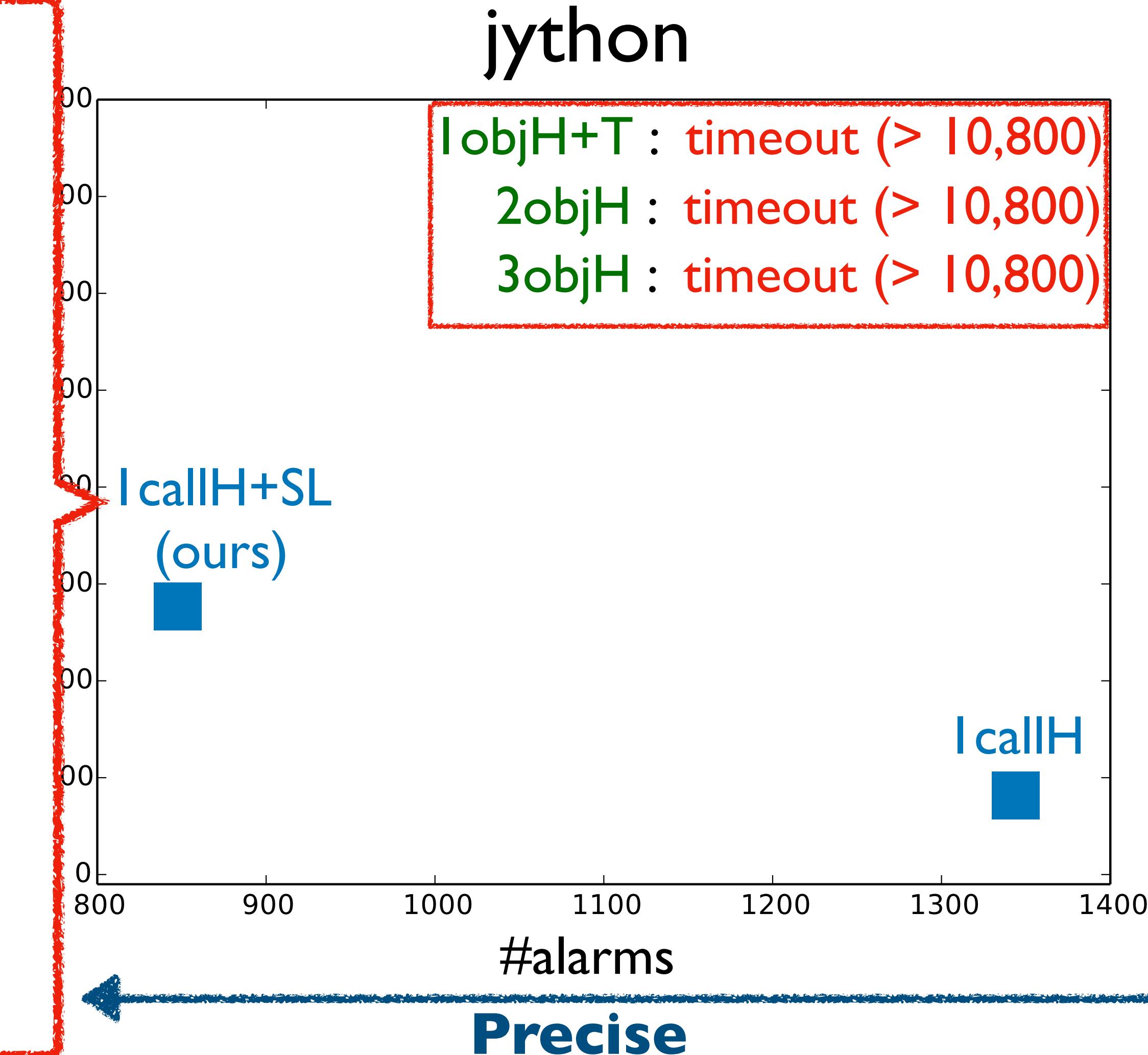
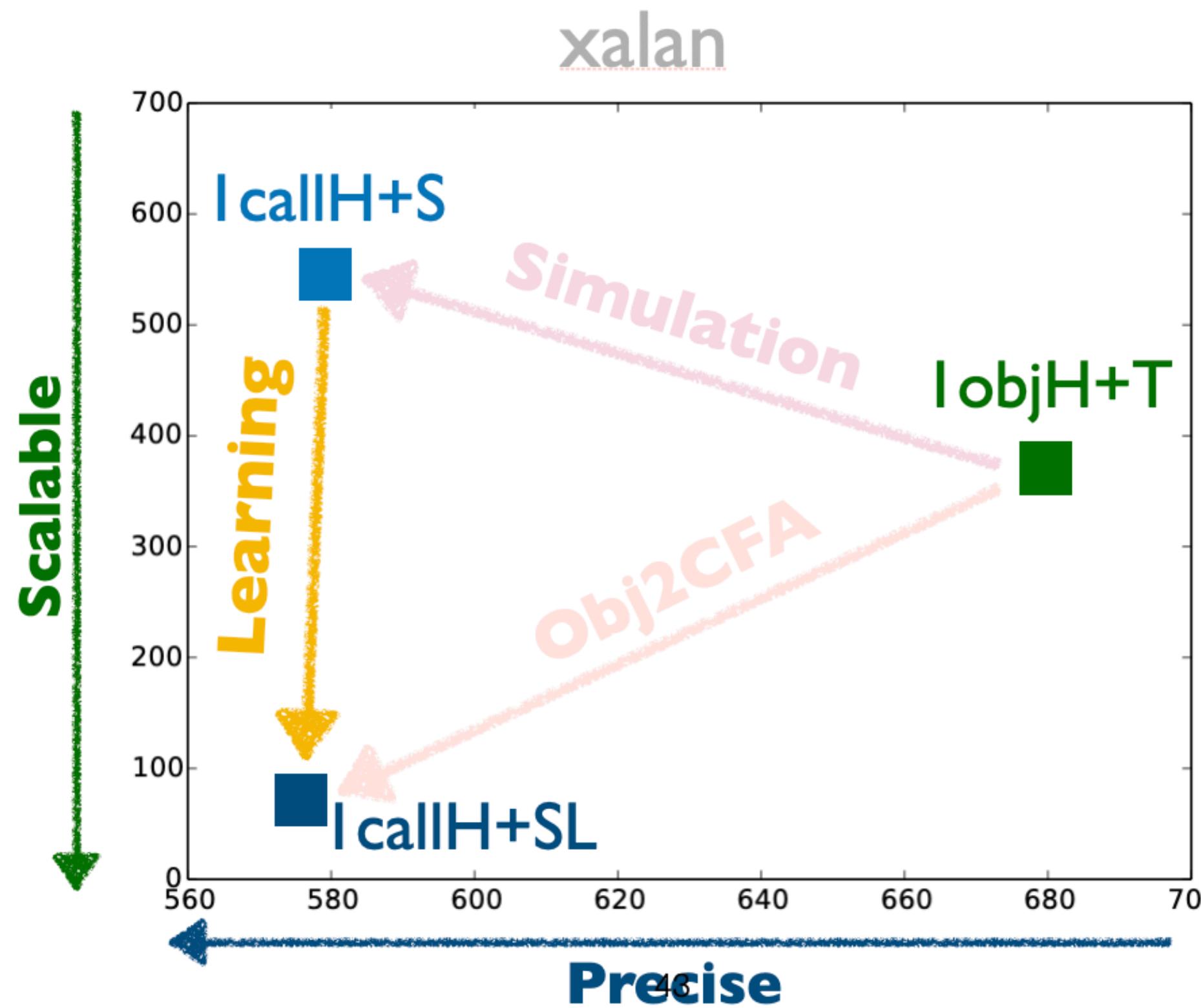
- $I_{callH+SL}$ (ours) is more precise and scalable than the existing object sensitivities



Call-site Sensitivity vs Object Sensitivity

- $I_{callH+SL}$ (ours) is more precise and scalable than the existing object sensitivities

- Necessity of learning
- $I_{callH+S}$ is unable to analyze jython



Summary

- Currently, CFA is known as a bad context
- However, if context tunneling is included, CFA is not a bad context anymore
- We need to reconsider CFA from now on

Thank you

Summary

- Currently, CFA is known as a bad context

- Call-site Sensitivity has been ignored

“... call-site-sensitivity is less important than others ...”
- Jeon et al. [2019]



1981

2002

2010

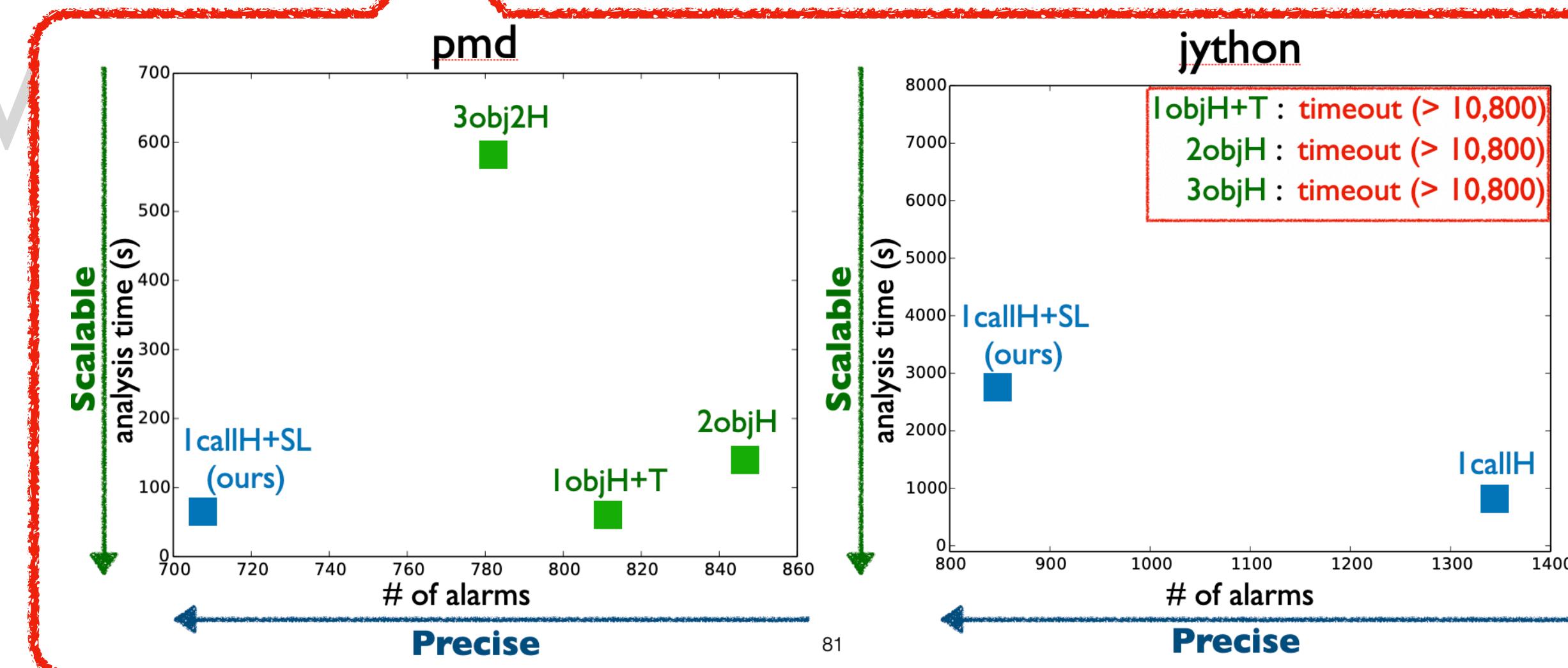
2022



Summary

- Currently, CFA is known as a bad context

• However, if context tunneling is included,
CFA is not a bad context anymore



• We can now on

Return of CFA: Call-Site Sensitivity Can Be Superior to Object Sensitivity Even for Object-Oriented Programs

MINSEOK JEON and HAKJOO OH*, Korea University, Republic of Korea

In this paper, we challenge the commonly-accepted wisdom in static analysis that object sensitivity is superior to call-site sensitivity for object-oriented programs. In static analysis of object-oriented programs, object sensitivity has been established as the dominant flavor of context sensitivity thanks to its outstanding precision. On the other hand, call-site sensitivity has been regarded as unsuitable and its use in practice has been constantly discouraged for object-oriented programs. In this paper, however, we claim that call-site sensitivity is generally a superior context abstraction because it is practically possible to transform object sensitivity into more precise call-site sensitivity. Our key insight is that the previously known superiority of object sensitivity holds only in the traditional k -limited setting, where the analysis is enforced to keep the most recent k context elements. However, it no longer holds in a recently-proposed, more general setting with context tunneling. With context tunneling, where the analysis is free to choose an arbitrary k -length subsequence of context strings, we show that call-site sensitivity can simulate object sensitivity almost completely, but not vice versa. To support the claim, we present a technique, called Obj2CFA, for transforming arbitrary context-tunneled object sensitivity into more precise, context-tunneled call-site-sensitivity. We implemented Obj2CFA in Doop and used it to derive a new call-site-sensitive analysis from a state-of-the-art object-sensitive pointer analysis. Experimental results confirm that the resulting call-site sensitivity outperforms object sensitivity in precision and scalability for real-world Java programs. Remarkably, our results show that even 1-call-site sensitivity can be more precise than the conventional 3-object-sensitive analysis.

1 INTRODUCTION

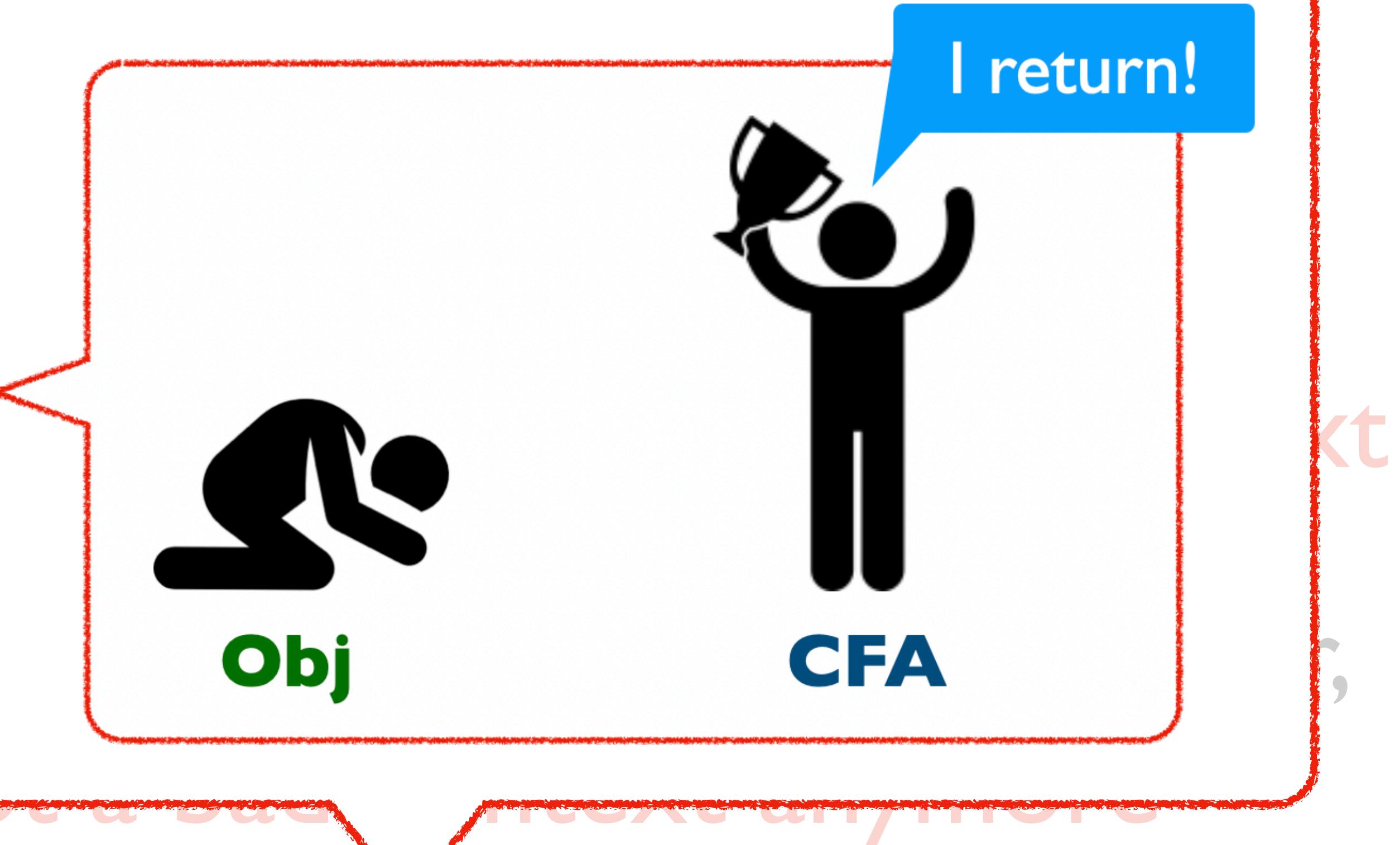
"Since its introduction, object sensitivity has emerged as the dominant flavor of context sensitivity for object-oriented languages."

—Smaragdakis and Balatsouras [2015]

Context sensitivity is critically important for static program analysis of object-oriented programs. A context-sensitive analysis associates local variables and heap objects with context information of method calls, computing analysis results separately for different contexts. This way, context sensitivity prevents analysis information from being merged along different call chains. For object-oriented and higher-order languages, it is well known that context sensitivity is the primary means

CFA wins!

uses the allocation-site of the receiver object (a) as the context of `foo`. The standard k -object-sensitive analysis [Milanova et al. 2002, 2005; Smaragdakis et al. 2011] maintains a sequence of



- We need to reconsider CFA from now on

Thank you