



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

Minseok Jeon and Hakjoo Oh



**KOREA**  
UNIVERSITY

POPL 2022 @ Philadelphia, USA



Two major camps

Call-Site Sensitivity

Object Sensitivity

Can  
Even for  
Object-Oriented Programs

Minseok Jeon and Hakjoo Oh



KOREA  
UNIVERSITY

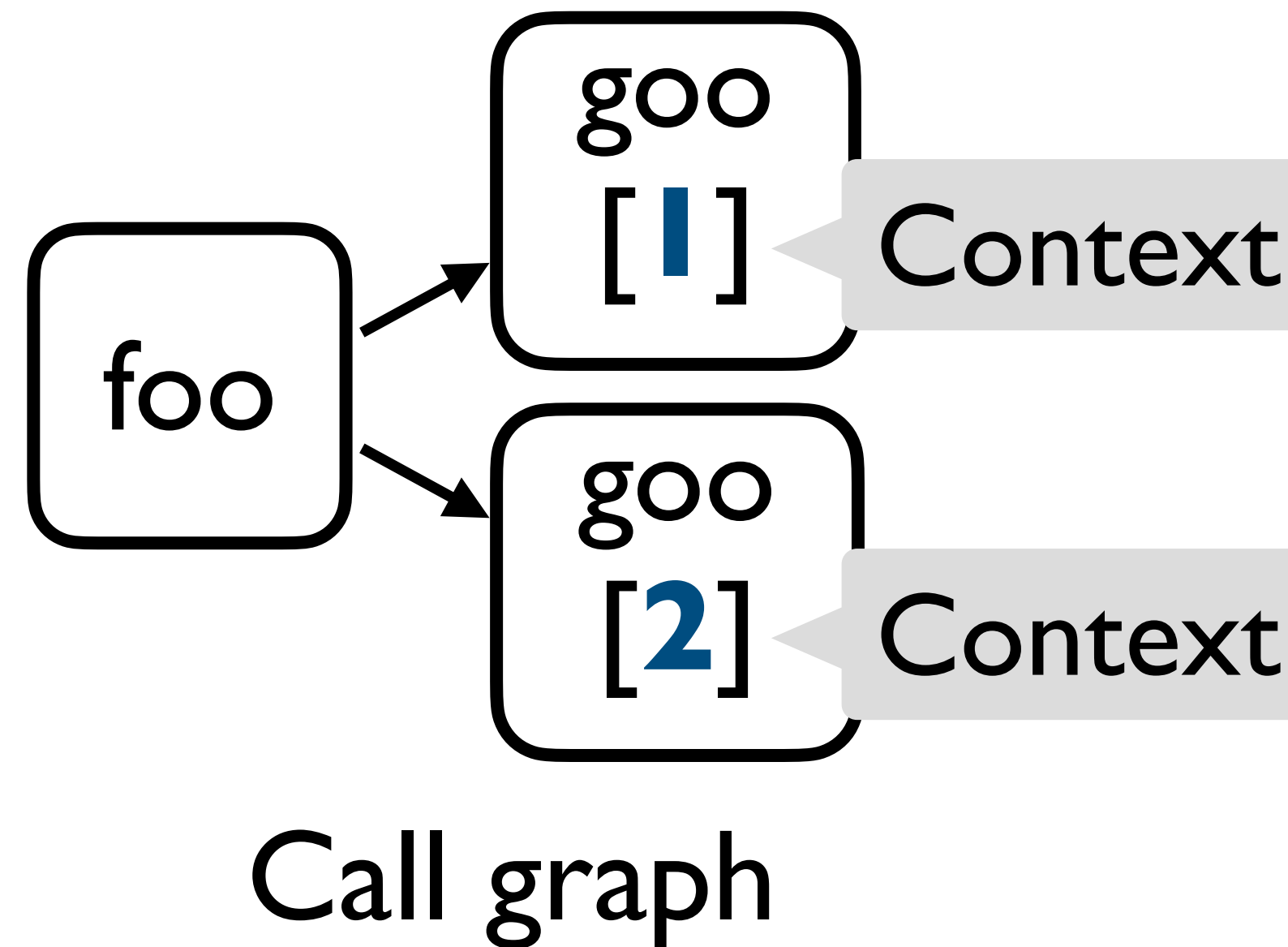
POPL 2022 @ Philadelphia, USA

# Call-site Sensitivity vs Object Sensitivity

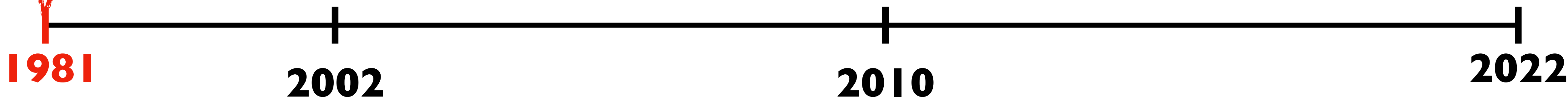
Call-site sensitivity was born in 1981

- Considers “**Where**”

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



Call-site sensitivity

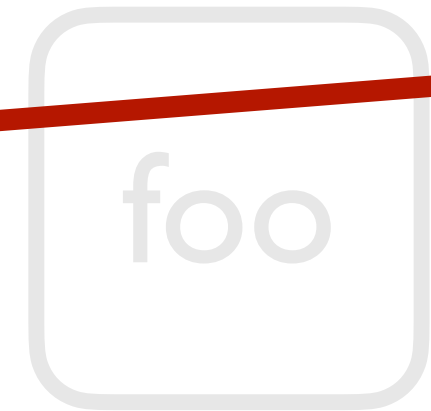


# Call-site Sensitivity vs Object Sensitivity

Call-site sensitivity was born in 1981

- Considers “**Where**”

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



Call graph

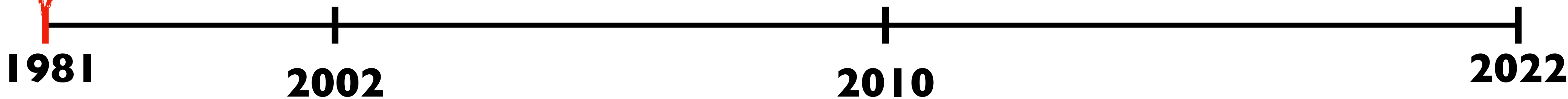
Call-site is context

Call-site is context

Where is it called from?



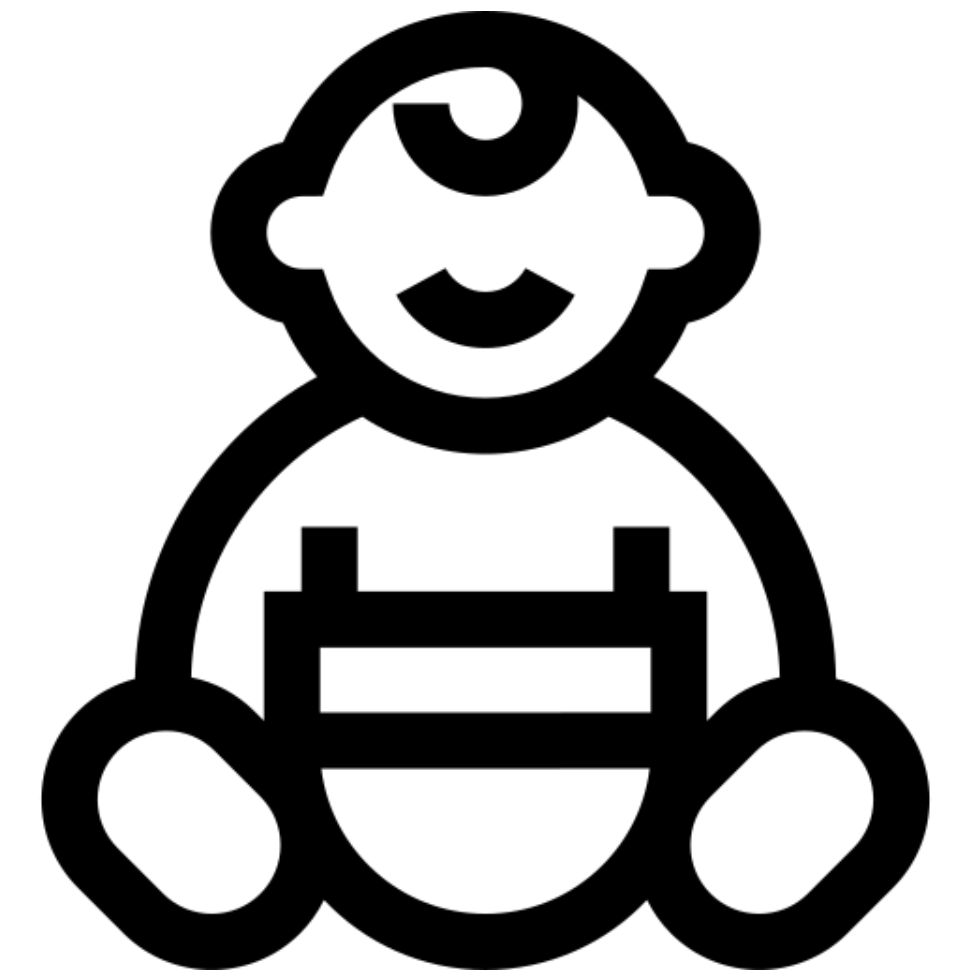
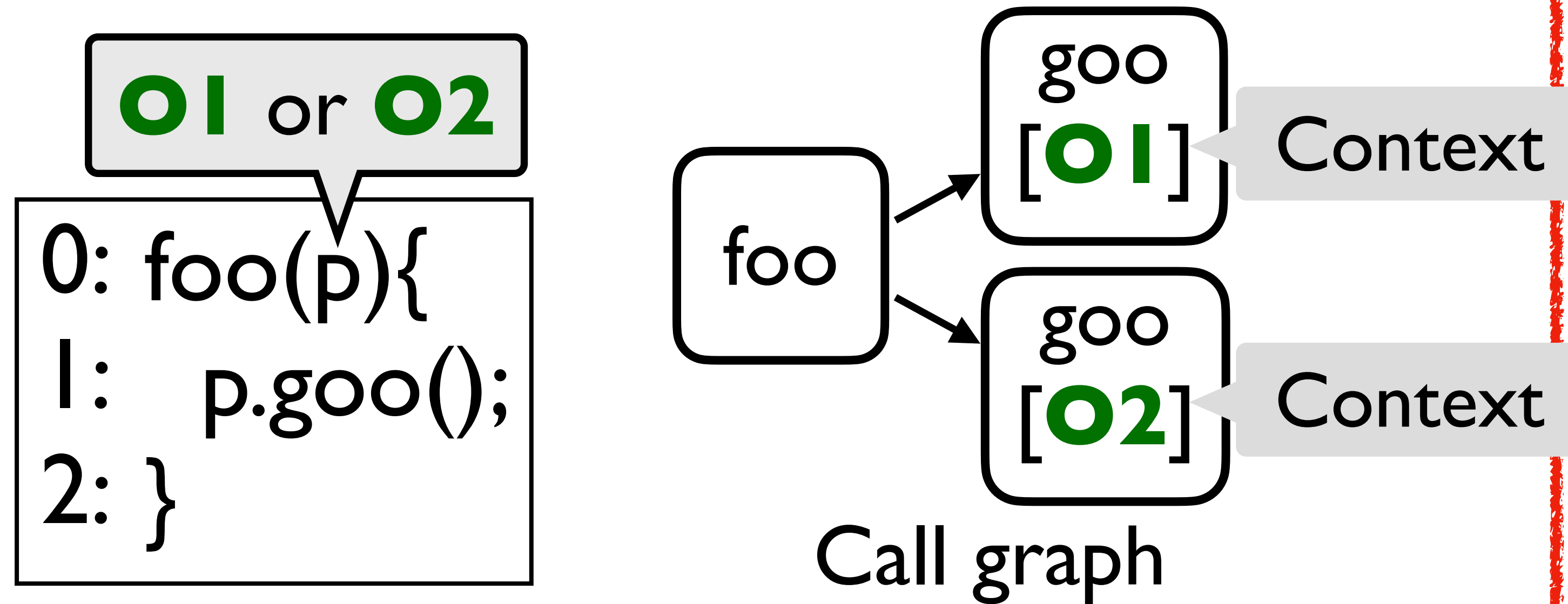
Call-site sensitivity



# Call-site Sensitivity vs Object Sensitivity

Object sensitivity appeared in 2002

- Considers “**What**”



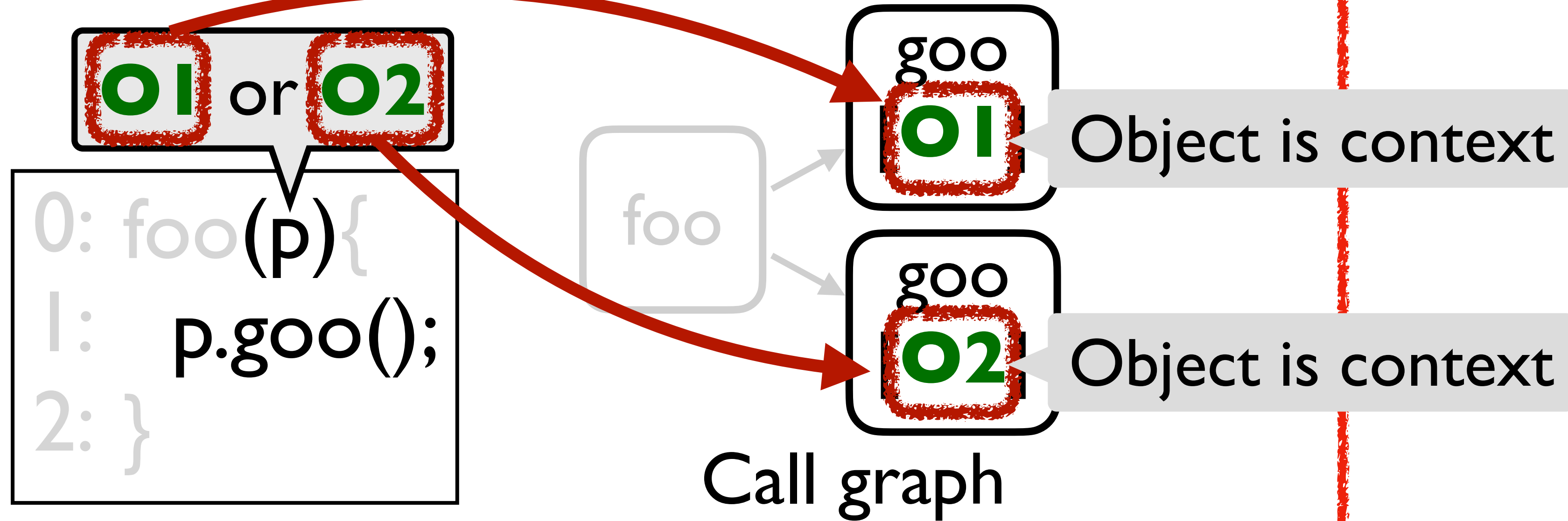
Object sensitivity



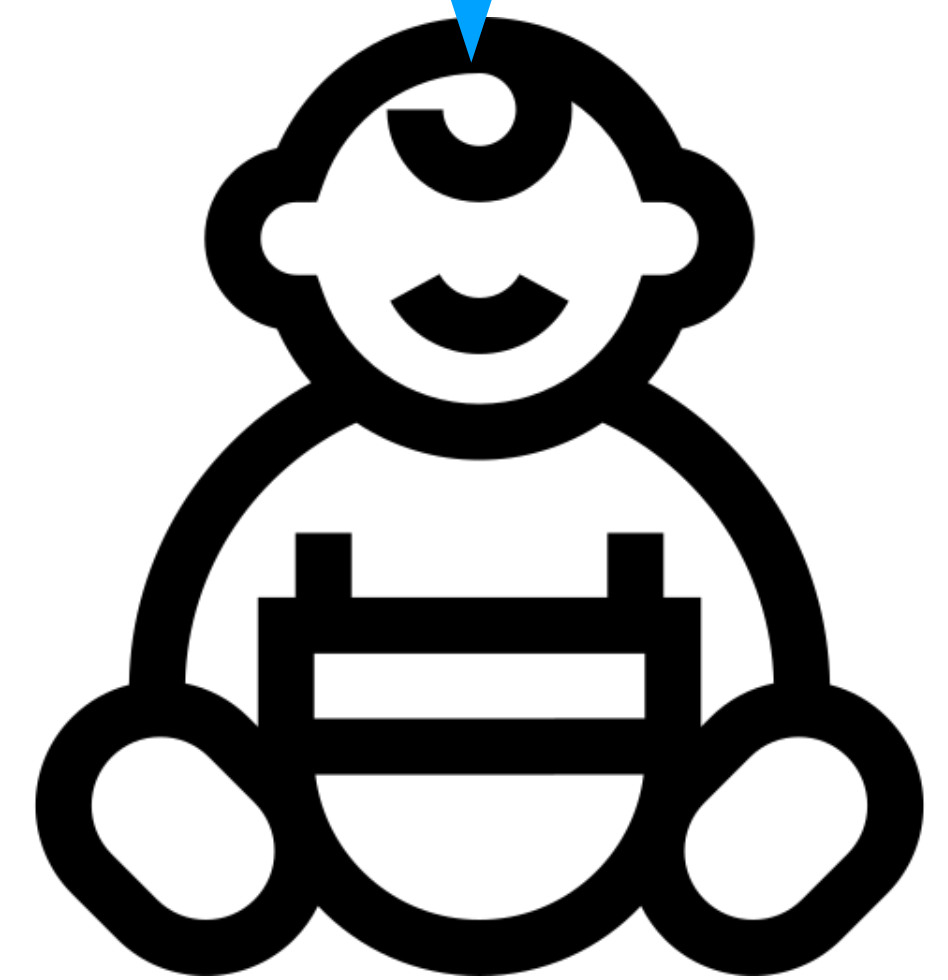
# Call-site Sensitivity vs Object Sensitivity

Object sensitivity appeared in 2002

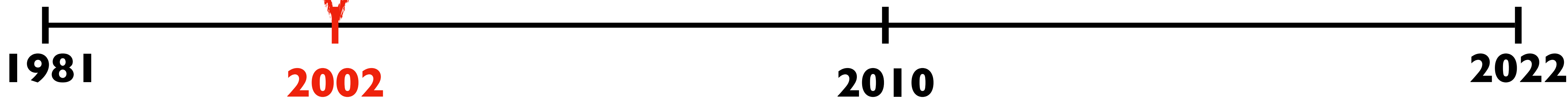
- Considers “**What**”



What is it called with?



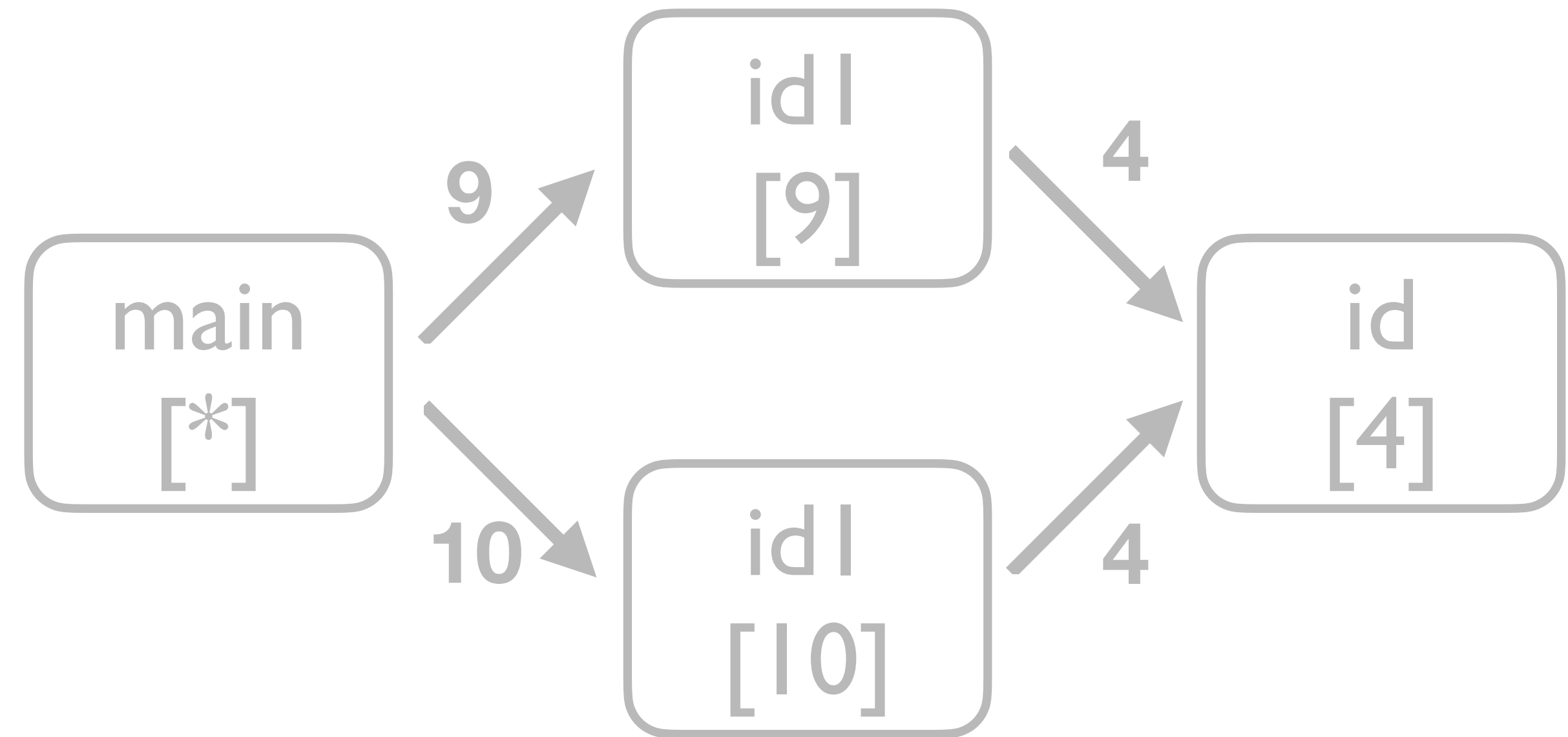
Object sensitivity



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



Call-graph of I-CFA

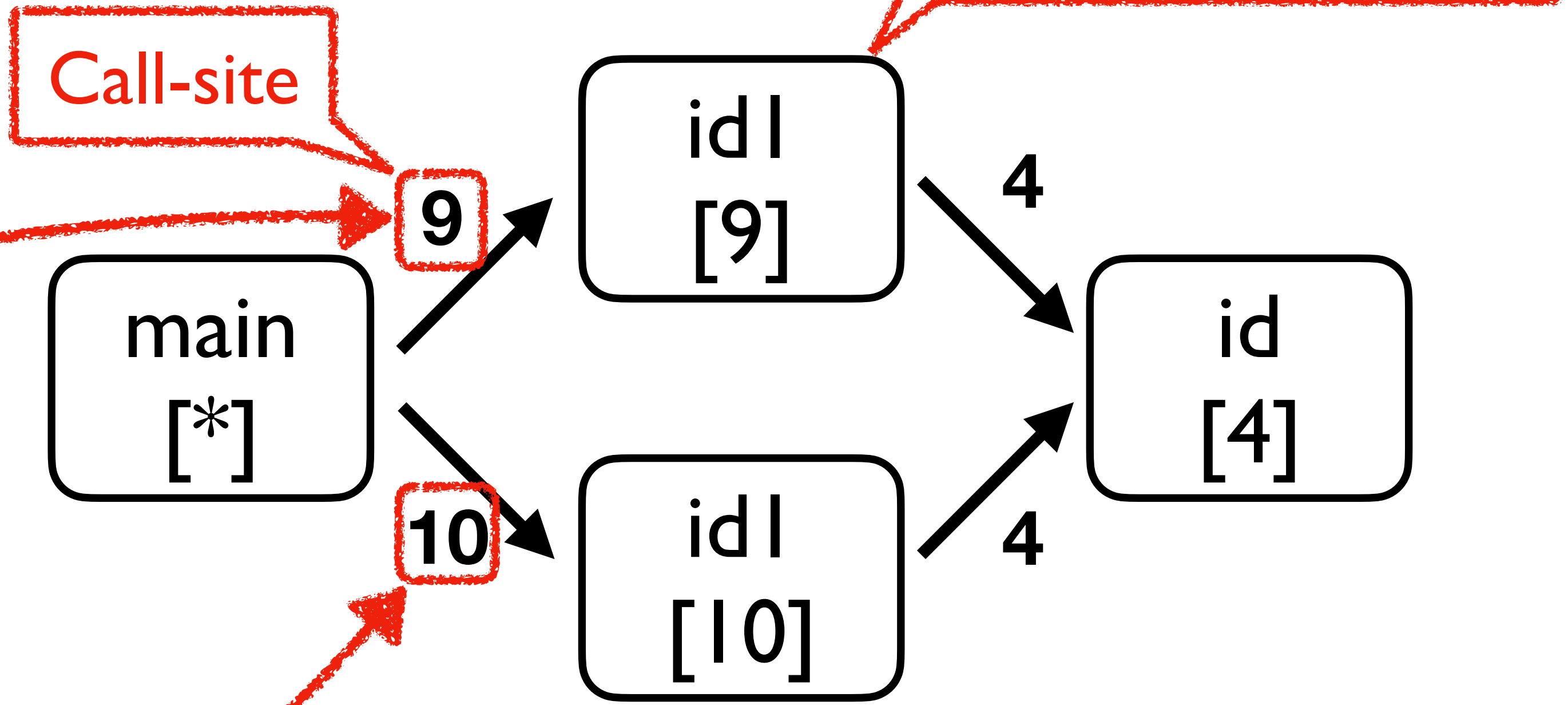
# Call-site Sensitivity vs Object Sensitivity

- An example shows the **limitation** of CFA and **strength** of **Method & Context**

```
0: class C{
1:   id(v){
2:     return v;}
3:   idl(v){
4:     return this.id(v);}
5: }
```

```
6: main(){
7:   c1 = new C();//C1
8:   c2 = new C();//C2
```

```
9:   a = (A) c1.idl(new A());//query1
10:  b = (B) c2.idl(new B());//query2
11: }
```



Call-graph of I-CFA

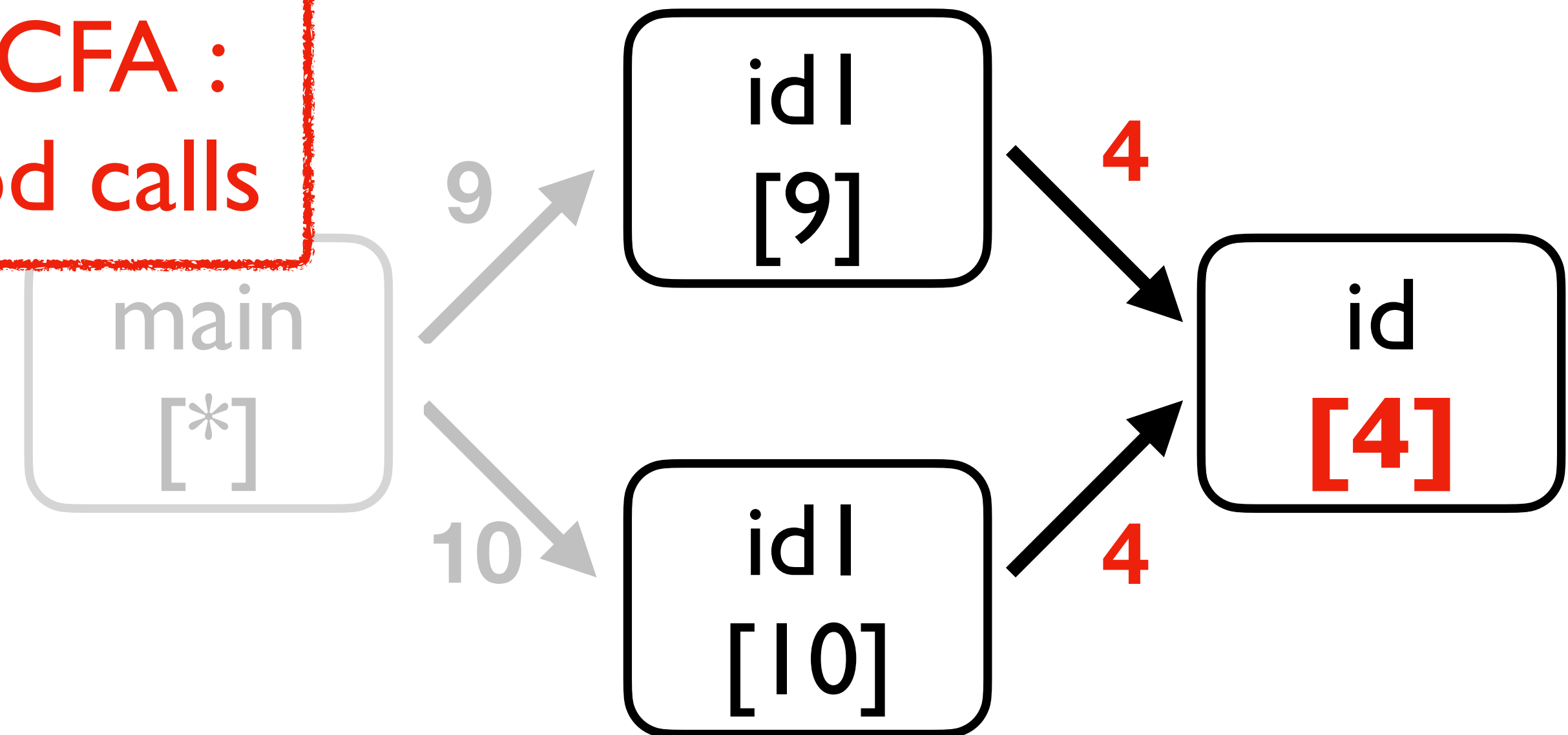


# Call-site Sensitivity vs Object Sensitivity

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

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

Limitation of CFA :  
Nested method calls

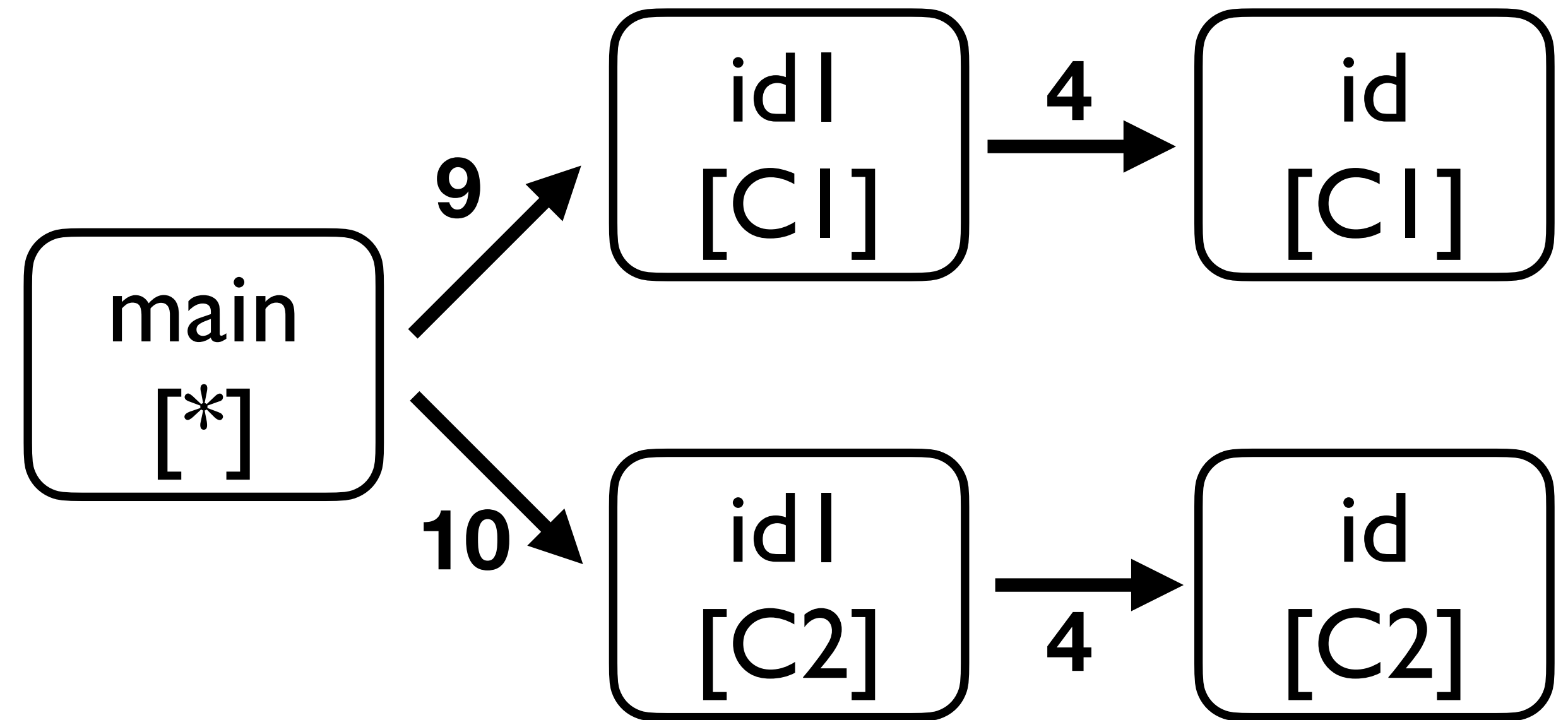


Call-graph of I-CFA

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



Call-graph of I-Obj

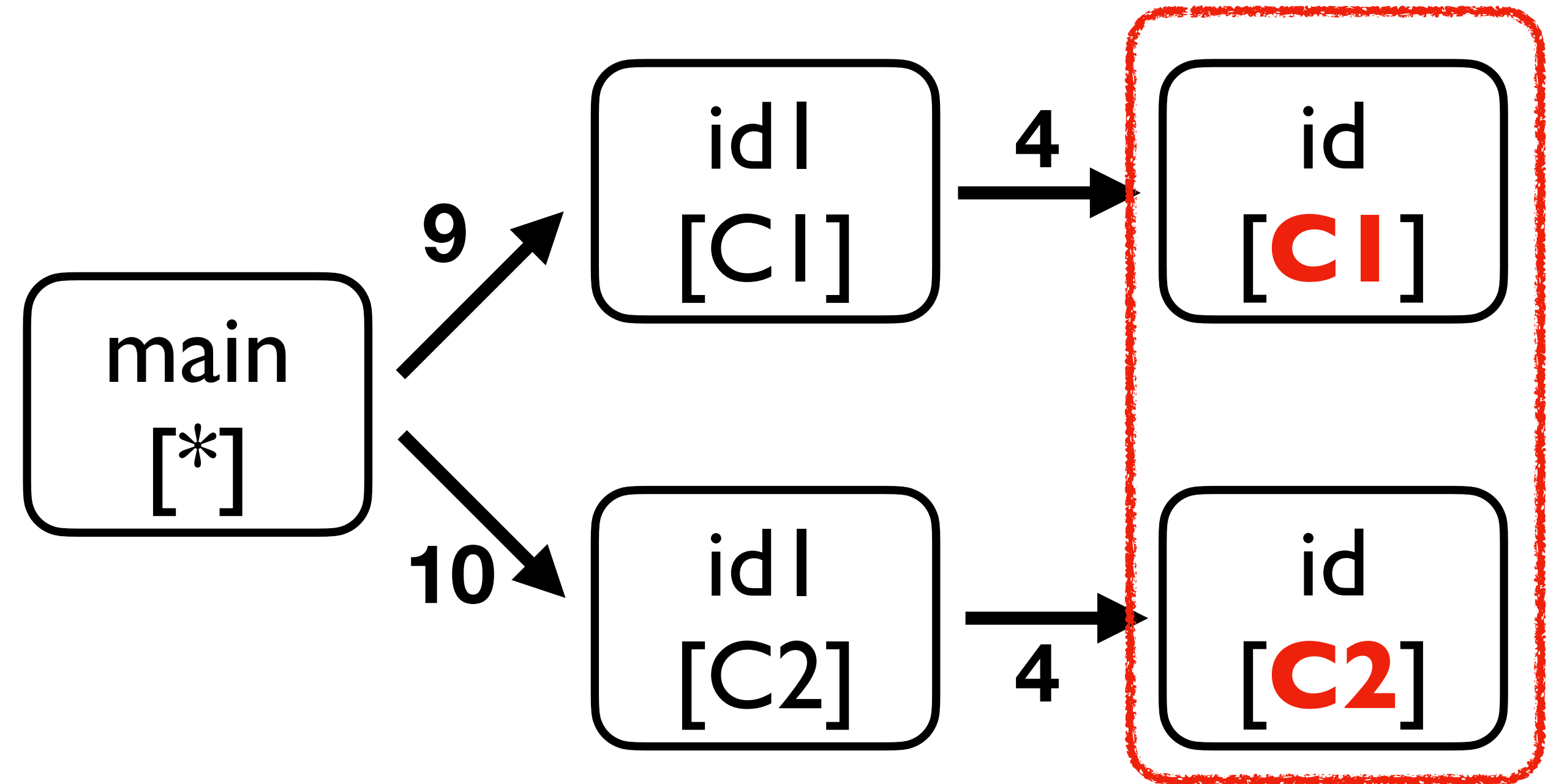
# Call-site Sensitivity vs Object Sensitivity

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

```
0: class C{
1:   id(v){
2:     return v;}
3:   idl(v){
4:     return this.id(v);}
5: }
6: main(){
```

C1 or C2

```
7:   c1 = new C();//C1
8:   c2 = new C();//C2
9:   a = (A) c1.idl(new A());//query1
10:  b = (B) c2.idl(new B());//query2
11: }
```

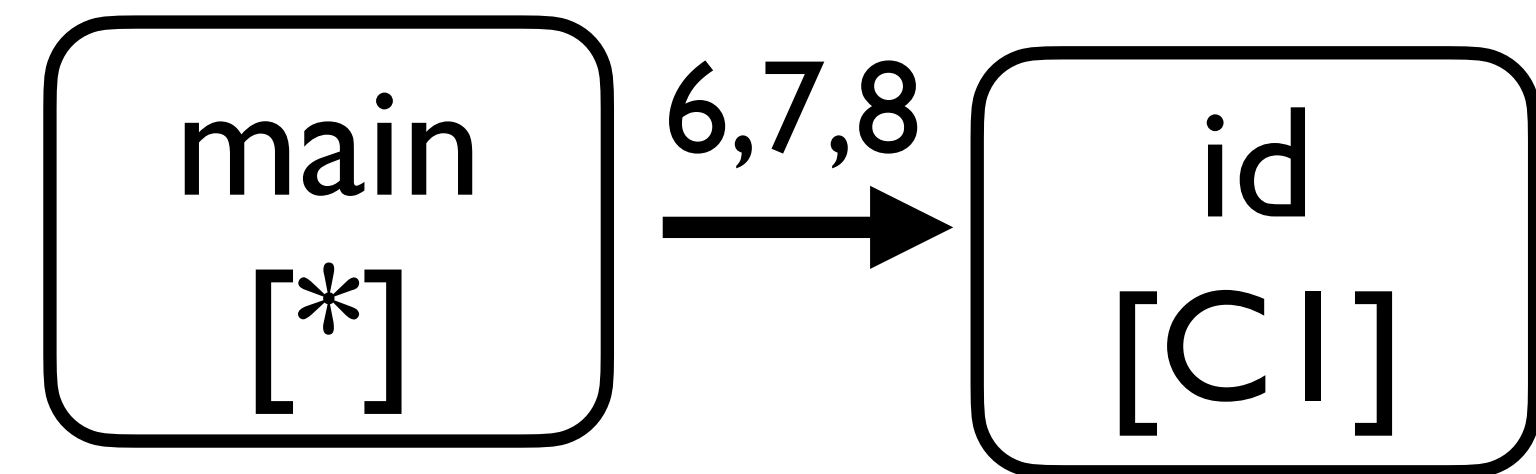


Call-graph of l-Obj

# 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:   c1 = new C();//CI
6:   a = (A) c1.id(new A());//query1
7:   b = (B) c1.id(new B());//query2
8:   c = (B) c1.id(new C());//query3
9: }
```

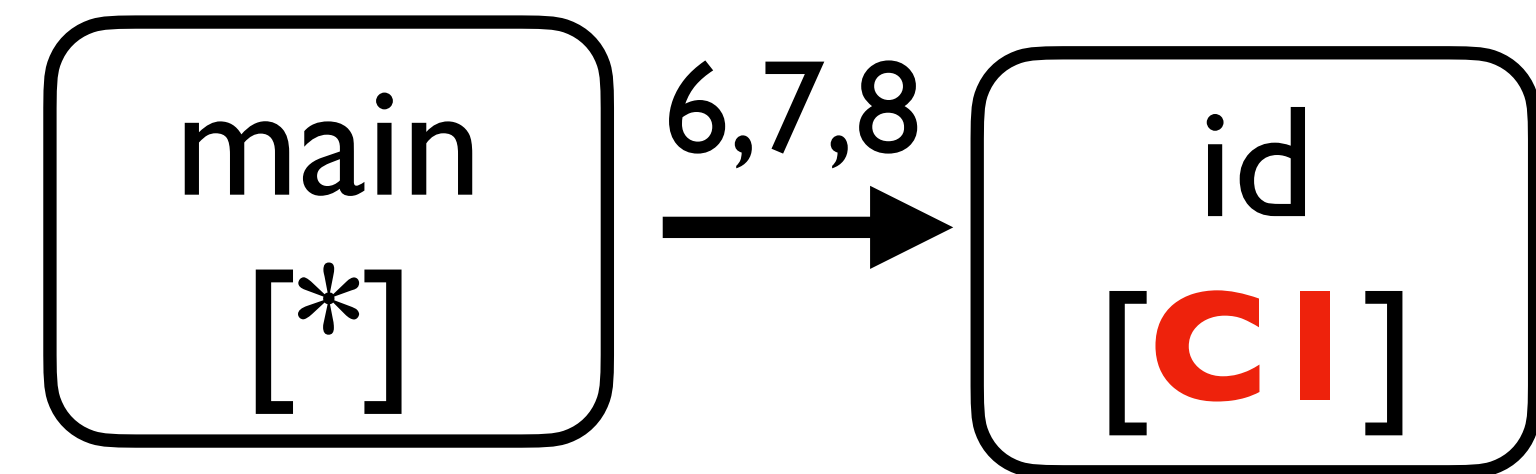


Call-graph of I-Obj

# 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:   cI = new C();//CI
6:   a = (A) cI.id(new A());//query1
7:   b = (B) cI.id(new B());//query2
8:   c = (B) cI.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

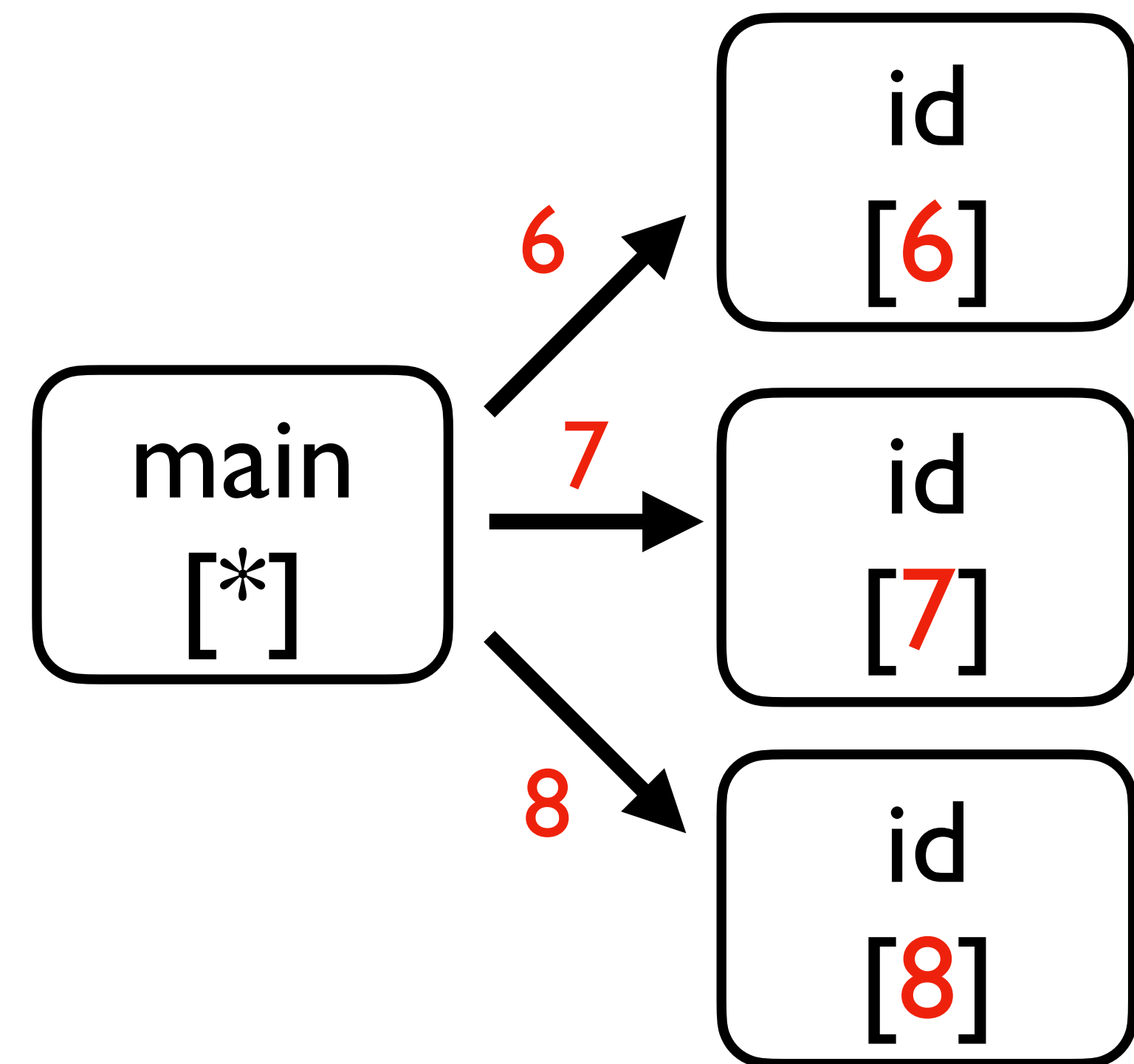
- An example shows the **limitation** of object sensitivity and **strength** of CFA

```
0: class C{
1:   id(v){
2:     return v;}
3: }
```

```
4: main(){
5:   c1 = new C();//C1
```

```
6:   a = (A) c1.id(new A());//query1
7:   b = (B) c1.id(new B());//query2
8:   c = (B) c1.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

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* determines the memory locations that may be modified by the execution of a program statement. *Def-use analysis* 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.

We have implemented two instantiations of our parameterized 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 for C. Our results also show that object sensitivity significantly improves the precision of side-effect analysis and call graph construction, compared to (1) context-insensitive analysis, and (2) context-sensitive points-to analysis that models context 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 (ISSTA)*, 2002, pp. 1-11.  
This research was supported in part by National Science Foundation (NSF) grant CCR-9900988. Author's addresses: A. Milanova, Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180; email: milanova@cs.rpi.edu; A. Rountev, Department of Computer Science and Engineering, Ohio State University, 2015 Neil Avenue, Columbus, OH 43210; email: rountev@ece.ohio-state.edu; B. G. Ryder, Department of Computer Science, Rutgers University, 100 Frelinghuysen Road, 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 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) 869-0481, or permission@acm.org.  
© 2005 ACM 1049-331X/05/1001-0001 \$5.00

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

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

Ondřej Lhoták<sup>1,2</sup> and Laurie Hendren<sup>2</sup>  
olhotak@uwaterloo.ca hendren@baile.mcgill.ca

<sup>1</sup> School of Computer Science, University of Waterloo, Waterloo, ON, Canada  
<sup>2</sup> School of Computer Science, McGill University, Montreal, QC, Canada

**Abstract.** We present the results of an empirical study evaluating the precision of subset-based points-to analysis with several variations of context sensitivity on Java benchmarks 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 contexts, the number of distinct contexts, and 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 devirtualizable 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 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.

#### 1 Introduction

Does context sensitivity significantly improve precision of interprocedural analysis of object-oriented programs? It is often suggested that it could, but lack of scalable implementations has hindered thorough empirical verification of this intuition.

Of the many context sensitive points-to analyses that have been proposed (e.g. [1, 4, 8, 11, 17-19, 25, 28-31]), which improve precision the most? Which are most effective for specific client analyses, and for specific code patterns? For which variations are we likely to find scalable implementations? Before devoting resources to finding efficient implementations of specific analyses, we should have empirical answers to these questions.

This study aims to provide these answers. Recent advances in the use of Binary Decision Diagrams (BDDs) in program analysis [3, 12, 29, 31] have made context sensitive analysis efficient enough to perform an empirical study on benchmarks of significant size. Using the JEDD system [14], we have implemented three different families of context-sensitive points-to analysis, and we have measured their precision in terms of several client analyses. Specifically, we compared the use of call-site strings as the context abstraction, object sensitivity [17, 18], and the algorithm proposed by Zhu and Calman [31]

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

### 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 *Passes*, a framework of BDD-based context-sensitive points-to and call graph analysis for Java, as well as client analyses that use their results. *Passes* 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 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.

As a result, *Passes* achieves several benefits, including full order-of-magnitude improvements in runtime. We compare *Passes* with Lhoták and Hendren's *Passes*, which defines the state of the art for context-sensitive analyses. For the exact same logical points-to definitions (and, consequently, identical precision) *Passes* is more than 15x faster than *Passes* for a 1-call-site sensitive analysis of the DaCapo benchmarks, with lower but still substantial speedups for other important analyses. Additionally, *Passes* scales to very precise analyses that are impossible with *Passes* and Whaley et al.'s *bdlddb*, 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 due to its declarativeness.

**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, Java, 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. Engin. 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., 2 Penn Plaza, Suite 701, New York, NY 10119-0709 USA, fax: +1 (212) 869-0481, or permission@acm.org.  
© 2008 ACM 1049-331X/08/09-ART3 \$5.00 DOI = 10.1145/1391984.1391987 <http://doi.acm.org/10.1145/1391984.1391987>

ACM Transactions on Software Engineering and Methodology, Vol. 18, No. 1, Article 3, Pub. date: September 2008.

### 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@acm.org yannis@cs.umass.edu

#### Abstract

We present the *Door* 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 defining (recursive) relations. We carry the declarative approach further than past work by describing the full end-to-end analysis in *Datalog* and optimizing aggressively using a novel technique specifically targeting highly recursive *Datalog* programs.

As a result, *Door* achieves several benefits, including full order-of-magnitude improvements in runtime. We compare *Door* with Lhoták and Hendren's *Passes*, which defines the state of the art for context-sensitive analyses. For the exact same logical points-to definitions (and, consequently, identical precision) *Door* is more than 15x faster than *Passes* for a 1-call-site sensitive analysis of the DaCapo benchmarks, with lower but still substantial speedups for other important analyses. Additionally, *Door* scales to very precise analyses that are impossible with *Passes* and Whaley et al.'s *bdlddb*, 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 due to its declarativeness.

**Categories and Subject Descriptors:** F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis; D.1.6 [Programming Techniques]: Logic Programming

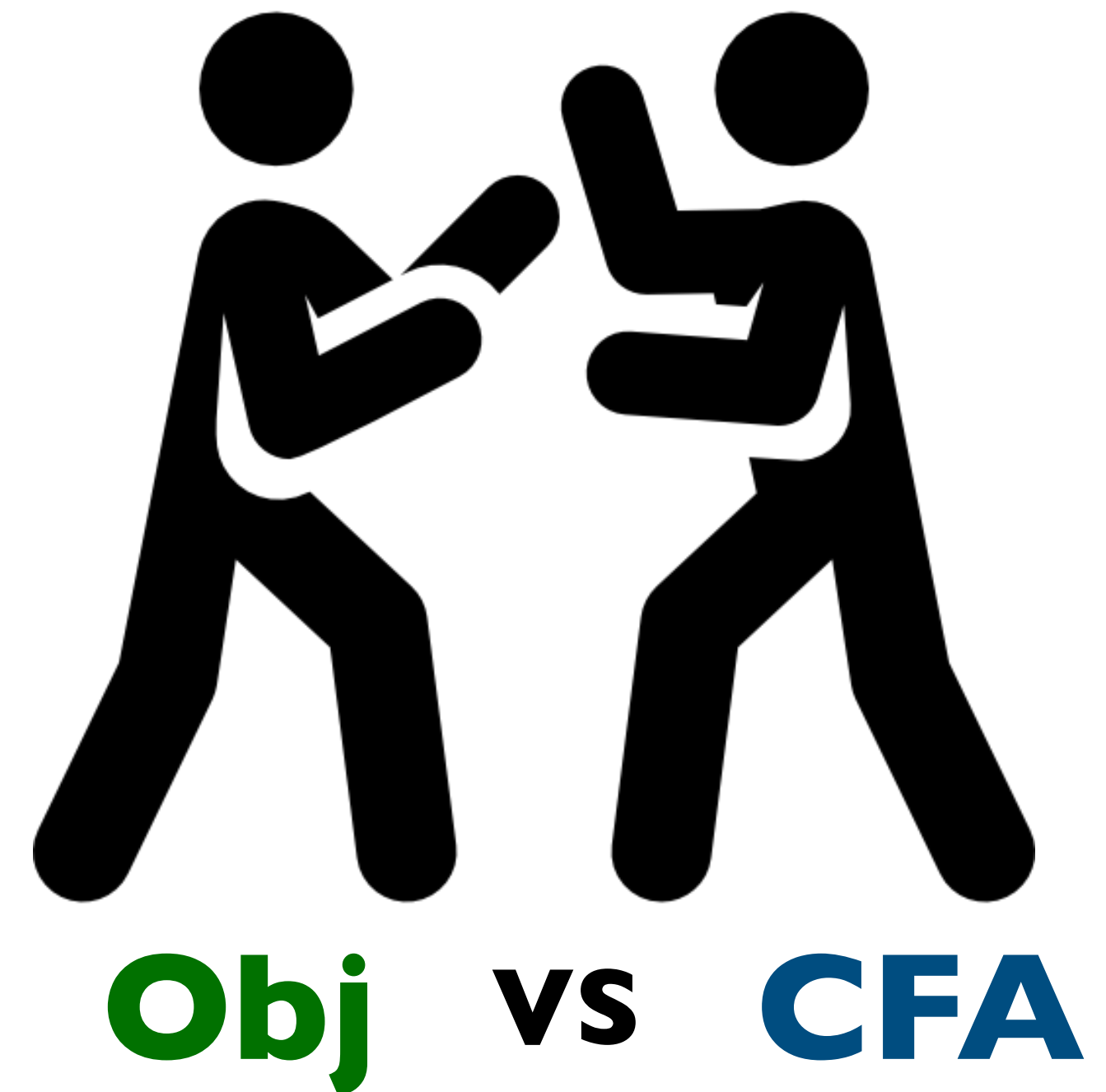
**General Terms:** Algorithms, Languages, Performance

#### 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 analyses. It is, thus, not surprising that a wealth of research has been devoted to efficient and precise pointer analysis techniques. Context-sensitive analyses are the most common class of precise 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 call-sites (for a *call-site sensitive analysis*—the traditional meaning of “context-sensitive”) or receiver objects (for an *object-sensitive analysis*).

In this work we present *Door*: a general and versatile points-to analysis framework that makes feasible the most precise context-sensitive analyses reported in the literature. *Door* implements a range of algorithms, including context insensitive, call-site sensitive, and object-sensitive analysis, all specified modularly as variations on a common code base. Compared to the prior state of the art, *Door* often achieves speedups of an order-of-magnitude for several important analyses.

The main elements of our approach are the use of the *Datalog* language for specifying the program analyses, and the aggressive optimization of 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: unoptimized analyses typically run over 1000 times more slowly. Generally 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, our approach is entirely *Datalog* based, encoding declaratively the logic required both for call graph construction as well as for handling the full semantic complexity of the Java language (e.g., static initialization, finalization, reference objects, threads, exceptions, reflection, etc.). This makes our pointer analysis specifications elegant, modular, but also 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



# 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 determines the memory locations that may be modified by the execution of a program statement. *Df* use analysis 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.

We have implemented two instantiations of our parameterized 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 for C. Our results also show that object sensitivity significantly improves the precision of side-effect analysis and call graph construction, compared to (1) context-insensitive analysis, and (2) context-sensitive points-to analysis that models context 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 (ISSTA)*, 2002, pp. 1-11.

This research was supported in part by National Science Foundation (NSF) grant CCR-9900988. Author's addresses: A. Milanova, Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180; email: milanova@cs.rpi.edu; A. Rountev, Department of Computer Science and Engineering, Ohio State University, 2015 Neil Avenue, Columbus, OH 43210; email: rountev@ce.ohio-state.edu; B. G. Ryder, Department of Computer Science, Rutgers University, 100 Frelinghuysen Road, 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

**Obj wins**

**Context-sensitive points-to analysis: is it worth it?\***  
 Ondřej Lhoták<sup>1,2</sup> and Laurie Hendren<sup>2</sup>  
 olhotak@uwaterloo.ca    hendren@saable.mcgill.ca  
<sup>1</sup> School of Computer Science, University of Waterloo, Waterloo, ON, Canada  
<sup>2</sup> School of Computer Science, McGill University, Montreal, QC, Canada

**Abstract.** We present the results of an empirical study evaluating the precision of subset-based points-to analysis with several variations of context sensitivity on Java benchmarks 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 contexts, the number of distinct contexts, and 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 deserializable 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 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.

**1 Introduction**

Does context sensitivity significantly improve precision of interprocedural analysis of object-oriented programs? It is often suggested that it could, but lack of scalable implementations has hindered thorough empirical verification of this intuition.

Of the many context sensitive points-to analyses that have been proposed (e.g. [1, 4, 8, 11, 17-19, 25, 28-31]), which improve precision the most? Which are most effective for specific client analyses, and for specific code patterns? For which variations are we likely to find scalable implementations? Before devoting resources to finding efficient implementations of specific analyses, we should have empirical answers to these questions.

This study aims to provide these answers. Recent advances in the use of Binary Decision Diagrams (BDDs) in program analysis [3, 12, 29, 31] have made context sensitive analysis efficient enough to perform an empirical study on benchmarks of significant size.

**Obj wins**

**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 *Passes*, a framework of BDD-based context-sensitive points-to and call graph analyses for Java, as well as client analyses that use their results. *Passes* 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 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.

As a result, *Door* achieves several benefits, including full order-of-magnitude improvements in runtime. We compare *Door* with Lhoták and Hendren's *Passes*, which defines the state of the art for context-sensitive analyses. For the exact same logical points-to definitions (and, consequently, identical precision) *Door* is more than 15x faster than *Passes* for a 1-call-site sensitive analysis of the *DuCapo* benchmarks, with lower but still substantial speedups for other important analyses. Additionally, *Door* scales to very precise analyses that are impossible with *Passes* and Whaley et al.'s *hblddb*, 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 due to its declarativeness.

**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, Java, 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. Engin. 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

**Obj wins**

**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@acm.org    yannis@cs.umass.edu

**Abstract**

We present the *Door* 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 defining (recursive) relations. We carry the declarative approach further than past work by describing the full end-to-end analysis in *Datalog* and optimizing aggressively using a novel technique specifically targeting highly recursive *Datalog* programs.

As a result, *Door* achieves several benefits, including full order-of-magnitude improvements in runtime. We compare *Door* with Lhoták and Hendren's *Passes*, which defines the state of the art for context-sensitive analyses. For the exact same logical points-to definitions (and, consequently, identical precision) *Door* is more than 15x faster than *Passes* for a 1-call-site sensitive analysis of the *DuCapo* benchmarks, with lower but still substantial speedups for other important analyses. Additionally, *Door* scales to very precise analyses that are impossible with *Passes* and Whaley et al.'s *hblddb*, 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 due to its declarativeness.

**Categories and Subject Descriptors:** E.3.2 [Logics and Meanings of Programs]: Semantics of 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 wins**





# Call-site Sensitivity vs Object Sensitivity

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



**Object-Sensitivity**

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

```
class A { void m() { return; } }
...
b = new B();
b.m();
```

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

Hakjoo Oh AAA616 2019 Fall, Lecture 8

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

```
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**

- Milanova, Rountev, and Ryder. *Parameterized sensitivity for points-to analysis for Java*. AC Eng. Methodol., 2005.
  - Context-sensitive interprocedural pointer analysis
  - For context, use stack of receiver objects
  - (More next week?)
- Lhotak and Hendren. *Context-sensitive pointer worth it?* CC 06
  - Object-sensitive pointer analysis more precise than for Java
  - 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 able to track the flow of information about the points-to set of `z`. If this information is available we can define the flow function as follows:

$$f_{CP[*p := v]}(\sigma) = [z \mapsto \sigma(z)]\sigma \text{ where } \text{must-point-to}(z, \sigma)$$

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

now the essence of knowledge Boston — Delft

**Pointer Analysis**

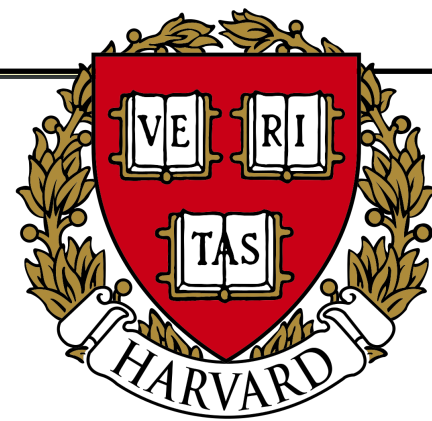
Yannis Smaragdakis  
University of Athens  
smaragd@di.uoa.gr

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

...



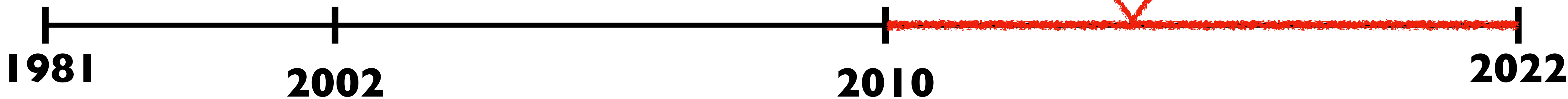
National and Kapodistrian University of Athens



Carnegie Mellon University



Obj



# 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 contexts, qualifying a method's local variables with the allocation site of the receiver object of the method call.

```
class A { void m() { return; } }  
...  
b = new B();  
b.m();  
The context of m is the allocation site of b.
```

Hakjoo Oh

AAA616 2019 Fall, Lecture 8

November 18, 2019

27 / 31



**KOREA**  
UNIVERSITY

National and Kapodistrian  
University of Athens



Carnegie  
Mellon  
University

**now**  
the essence of knowledge

I was also taught like that



**Obj**

1981

2002

2010

2022

# Call-site Sensitivity vs Object Sensitivity

- Researches focused on improving Object Sensitivity

## Researches on Object Sensitivity

**Pick Your Call-Site Sensitivity**  
Yannis Smaragdakis, University of Massachusetts Amherst, MA 01003, USA and Department of Informatics, University of Athens, 15701 Athens, Greece  
Abstract: Object sensitivity has emerged as a key technique for pointer analysis. However, object sensitivity is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

**Hybrid Context-Sensitive Pointer Analysis**  
Tian Tan<sup>1</sup>, Yue Li<sup>2</sup>, School of Computer Science and Technology, Tsinghua University, Beijing 100084, China  
Abstract: Object-sensitivity abstraction for pointer analysis is a key technique for analyzing heap memory. However, object-sensitivity abstraction is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

**Making  $k$ -Object-Sensitive Pointer Analysis More Precise with Still  $k$ -Limiting**  
Tian Tan<sup>1</sup>, Yue Li<sup>2</sup>, School of Computer Science and Technology, Tsinghua University, Beijing 100084, China  
Abstract: Object-sensitivity abstraction for pointer analysis is a key technique for analyzing heap memory. However, object-sensitivity abstraction is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

**Efficient Modeling the Heap**  
Yue Li, Tian Tan, Anders Møller, and Yann Smaragdakis, University of Copenhagen, Copenhagen, Denmark  
Abstract: Object-sensitivity abstraction for pointer analysis is a key technique for analyzing heap memory. However, object-sensitivity abstraction is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

**Precision-Guided Context-Sensitive Pointer Analysis**  
Yue Li, Tian Tan, Anders Møller, and Yann Smaragdakis, University of Copenhagen, Copenhagen, Denmark  
Abstract: Object-sensitivity abstraction for pointer analysis is a key technique for analyzing heap memory. However, object-sensitivity abstraction is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

**Scalability-First Self-Tuning Context-Sensitive Pointer Analysis**  
Yue Li, Tian Tan, Aarhus University, Denmark  
Abstract: Object-sensitivity abstraction for pointer analysis is a key technique for analyzing heap memory. However, object-sensitivity abstraction is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

**Data-Driven Context-Sensitive Pointer Analysis**  
Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh<sup>\*</sup>, Korea University, Seoul, Korea  
Abstract: Object-sensitivity abstraction for pointer analysis is a key technique for analyzing heap memory. However, object-sensitivity abstraction is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

**Learning Graph-based Heuristics without Handcrafting Application-Specific Features**  
Minseok Jeon, MyungHo Lee, and Hakjoo Oh<sup>\*</sup>, Korea University, Seoul, Korea  
Abstract: Object-sensitivity abstraction for pointer analysis is a key technique for analyzing heap memory. However, object-sensitivity abstraction is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

**Precision-Preserving Pointer Analysis with Partial Context-Sensitivity**  
Jingbo Lu, UNSW Sydney, Australia  
Abstract: Object-sensitivity abstraction for pointer analysis is a key technique for analyzing heap memory. However, object-sensitivity abstraction is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

**Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity**  
Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis, University of Athens, Greece  
Abstract: Object-sensitivity abstraction for pointer analysis is a key technique for analyzing heap memory. However, object-sensitivity abstraction is often limited by the precision of the underlying pointer analysis. In this paper, we propose a new technique for automatically learning graph-based heuristics for object sensitivity. We present a new algorithm for learning graph-based heuristics for object sensitivity. We show that our approach is competitive with the existing state-of-the-art.

198

2002

2010

2022



# 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

**A Machine-Learning Approach to Data-Driven Program Analysis**  
MINSEOK JEON, SEHUN JEONG, Republic of Korea  
We present a new machine-learning approach to data-driven program analysis. One major challenge in static program analysis is the analysis performance. Recently, data-driven approaches have been promising for various tasks, such as program debugging and code deobfuscation, program properties, and so on. However, for data-driven program analysis as well as for other tasks, the analysis performance is often bottlenecked by the analysis time. In this paper, we propose a new machine-learning approach to data-driven program analysis. We show that our automated technique significantly outperforms the state-of-the-art techniques, including ones hand-crafted by human experts.

**1 INTRODUCTION**  
One major challenge in static program analysis is a substantial amount of manual effort for tuning the analysis performance for real-world applications. Practical static analysis tools often use various heuristics to optimize their performance. For example, context-sensitive analysis for object-oriented programs, as it distinguishes method's local variables and different calling contexts. However, applying context-sensitive methods to all methods does not scale and therefore real-world static analyzers apply context-sensitive methods determined by some heuristic rules [Smaragdakis et al. 2014]. Another relational analysis such as ones with Octagons [Jitoe 2006]. Because it is impractical of all variable relationships in the program, static analyzers employ variable-chains

**1 INTRODUCTION**  
Pointer analysis, as an enabling technology, plays a key role in a wide range of applications, including bug detection [2, 25, 35, 34], security analysis [6, 33], and program understanding [12]. The main goal of pointer analysis is to track the flow of pointers in a program. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11]. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11]. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11].

**1 INTRODUCTION**  
Pointer analysis, as an enabling technology, plays a key role in a wide range of applications, including bug detection [2, 25, 35, 34], security analysis [6, 33], and program understanding [12]. The main goal of pointer analysis is to track the flow of pointers in a program. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11]. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11].

**1 INTRODUCTION**  
Pointer analysis, as an enabling technology, plays a key role in a wide range of applications, including bug detection [2, 25, 35, 34], security analysis [6, 33], and program understanding [12]. The main goal of pointer analysis is to track the flow of pointers in a program. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11]. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11].

**1 INTRODUCTION**  
Pointer analysis, as an enabling technology, plays a key role in a wide range of applications, including bug detection [2, 25, 35, 34], security analysis [6, 33], and program understanding [12]. The main goal of pointer analysis is to track the flow of pointers in a program. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11]. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11].

**1 INTRODUCTION**  
Pointer analysis, as an enabling technology, plays a key role in a wide range of applications, including bug detection [2, 25, 35, 34], security analysis [6, 33], and program understanding [12]. The main goal of pointer analysis is to track the flow of pointers in a program. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11]. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11].

**1 INTRODUCTION**  
Pointer analysis, as an enabling technology, plays a key role in a wide range of applications, including bug detection [2, 25, 35, 34], security analysis [6, 33], and program understanding [12]. The main goal of pointer analysis is to track the flow of pointers in a program. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11]. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11].

**1 INTRODUCTION**  
Pointer analysis, as an enabling technology, plays a key role in a wide range of applications, including bug detection [2, 25, 35, 34], security analysis [6, 33], and program understanding [12]. The main goal of pointer analysis is to track the flow of pointers in a program. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11]. For object-oriented programs, e.g., Java programs, however, context-sensitive pointer analysis is needed by many clients [11].

1981

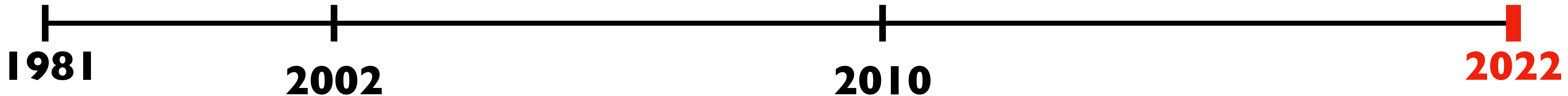
2002

2010

2022

# 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

## Precise and Scalable Points-to Analysis via Data-Driven Context Tunneling

MINSEOK JEON, Korea University, Republic of Korea  
SEHUN JEONG, Korea University, Republic of Korea  
HAKJOO OH\*, Korea University, Republic of Korea

We present context tunneling, a new approach for making  $k$ -limited context-sensitive points-to analysis precise and scalable. As context-sensitivity holds the key to the development of precise and scalable points-to analysis, a variety of techniques for context-sensitivity have been proposed. However, existing approaches such as  $k$ -call-site-sensitivity or  $k$ -object-sensitivity have a significant weakness that they unconditionally update the context of a method at every call-site, allowing important context elements to be overwritten by more recent, but not necessarily more important, context elements. In this paper, we show that this is a key limiting factor of existing context-sensitive analyses, and demonstrate that remarkable increase in both precision and scalability can be gained by maintaining important context elements only. Our approach, called context tunneling, updates contexts selectively and decides when to propagate the same context without modification.

We attain context tunneling via a data-driven approach. The effectiveness of context tunneling is very sensitive to the choice of important context elements. Even worse, precision is not monotonically increasing with respect to the ordering of the choices. As a result, manually coming up with a good heuristic rule for context tunneling is extremely challenging and likely fails to maximize its potential. We address this challenge by developing a specialized data-driven algorithm, which is able to automatically search for high-quality heuristics over the non-monotonic space of context tunneling.

We implemented our approach in the Doop framework and applied it to four major flavors of context-sensitivity: call-site-sensitivity, object-sensitivity, type-sensitivity, and hybrid context-sensitivity. In all cases, 1-context-sensitive analysis with context tunneling far outperformed deeper context-sensitivity with  $k = 2$  in both precision and scalability.

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

Additional Key Words and Phrases: Points-to analysis, Context-sensitive analysis, Data-driven program analysis

### ACM Reference Format:

Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and Scalable Points-to Analysis via Data-Driven Context Tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 140 (November 2018), 30 pages. <https://doi.org/10.1145/3276510>

\*Corresponding author

Authors' addresses: Minseok Jeon, [minseok.jeon@korea.ac.kr](mailto:minseok.jeon@korea.ac.kr), Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea; Sehun Jeong, [gifaranga@korea.ac.kr](mailto:gifaranga@korea.ac.kr), Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea; Hakjoo Oh, [hakjoo\\_oh@korea.ac.kr](mailto:hakjoo_oh@korea.ac.kr), Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2018 Association for Computing Machinery.  
2475-1421/2018/11-ART140  
<https://doi.org/10.1145/3276510>

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

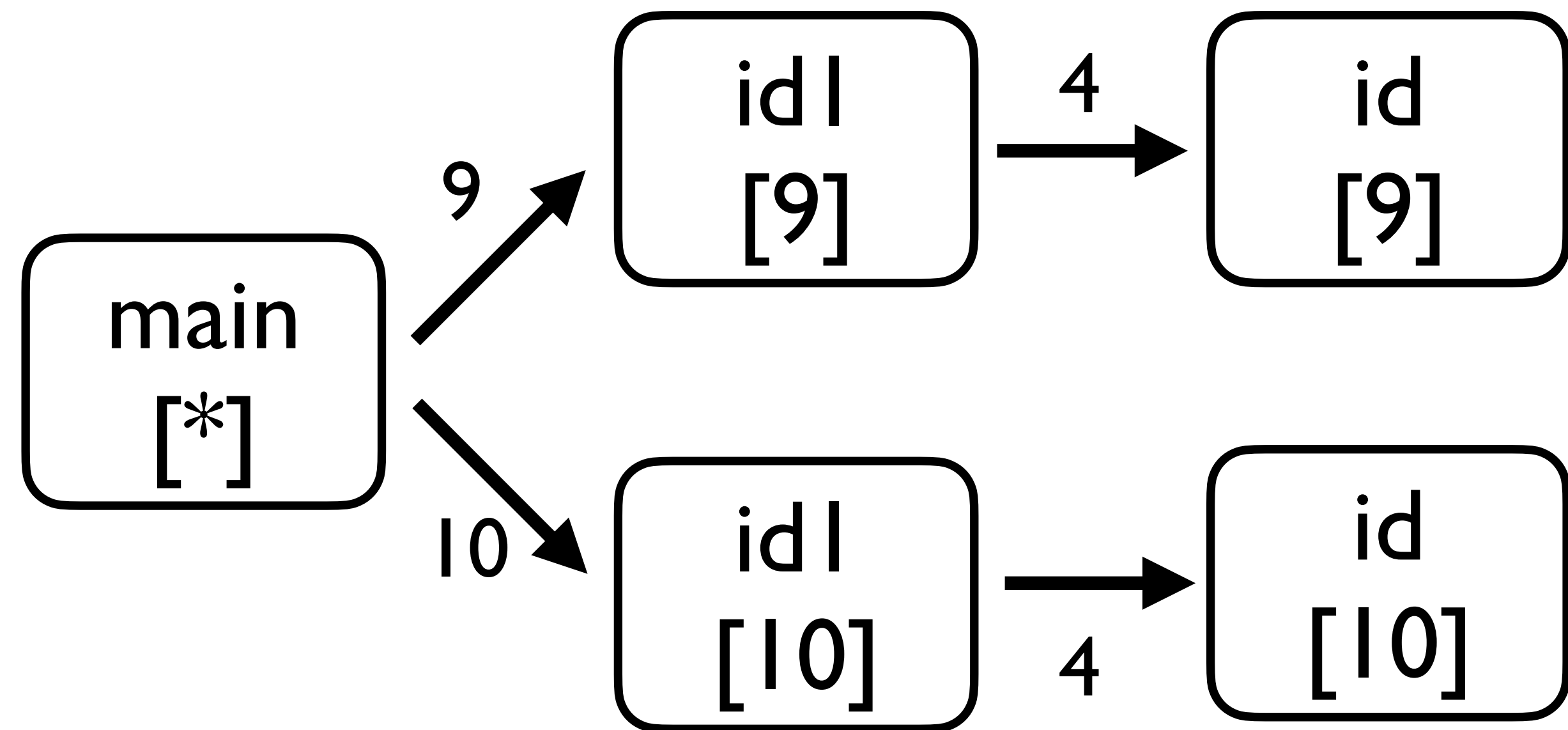
Jeon et al. [2018]



# 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:   idl(v){
4:     return id0(v);}
5: }
6: main(){
7:   c1 = new C();//C1
8:   c2 = new C();//C2
9:   a = (A) c1.idl(new A());//query1
10:  b = (B) c2.idl(new B());//query2
11: }
```



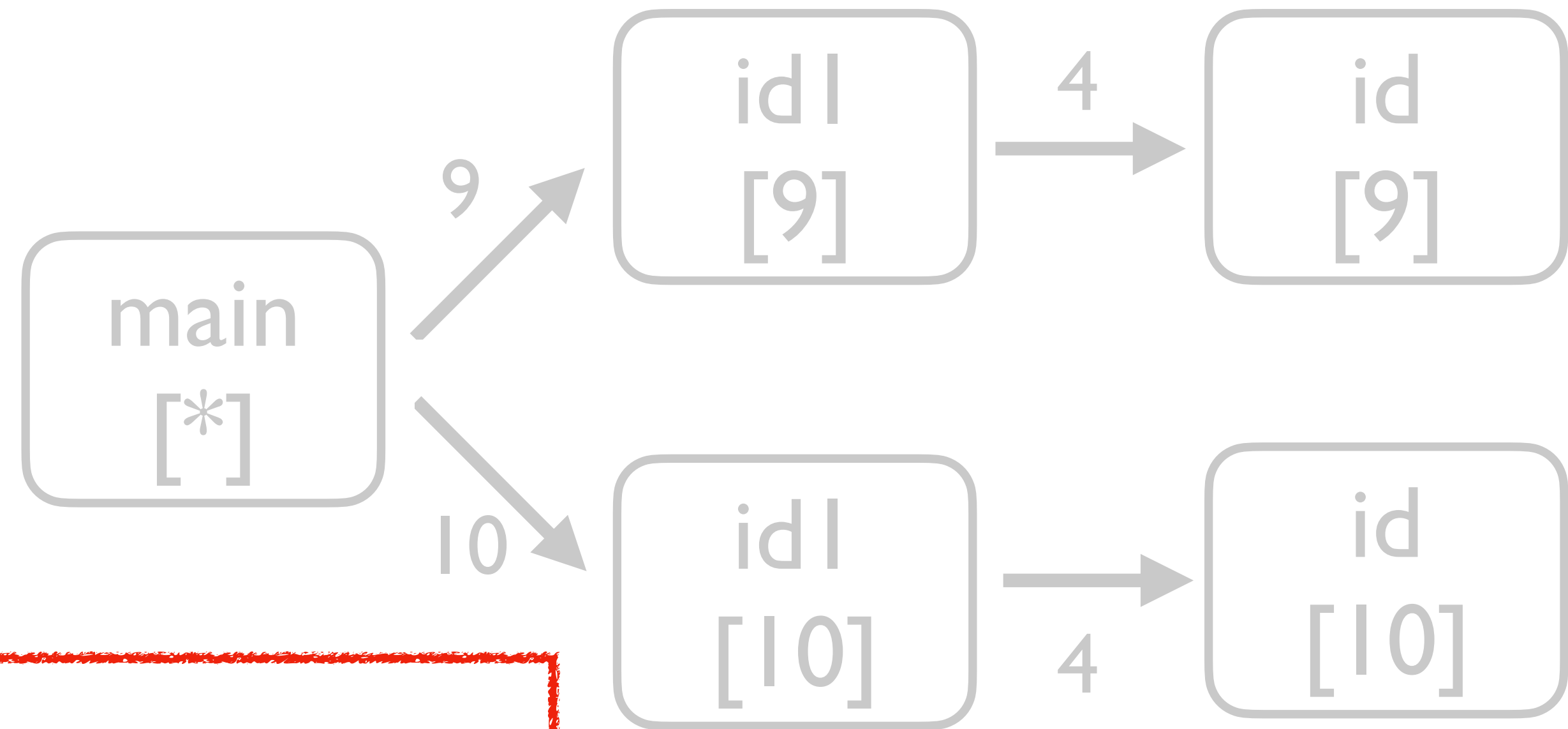
I-CFA with context tunneling  
( $T = \{4\}$ )



# 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:   idl(v){
4:     return id0(v);}
5: }
6: main(){
7:   c1 = new C();//C1
```



**Tunneling abstraction:**

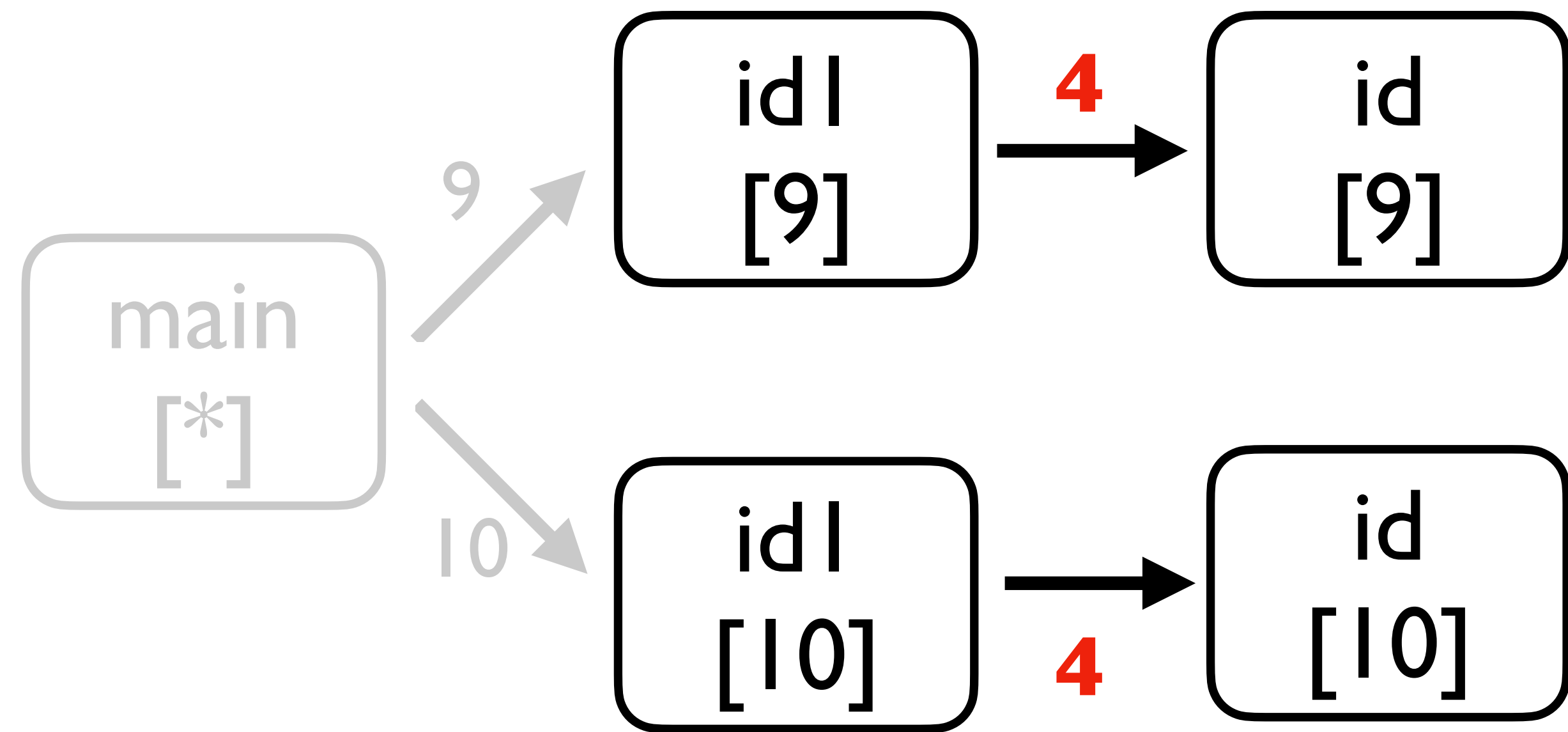
Determines where to apply context tunneling

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



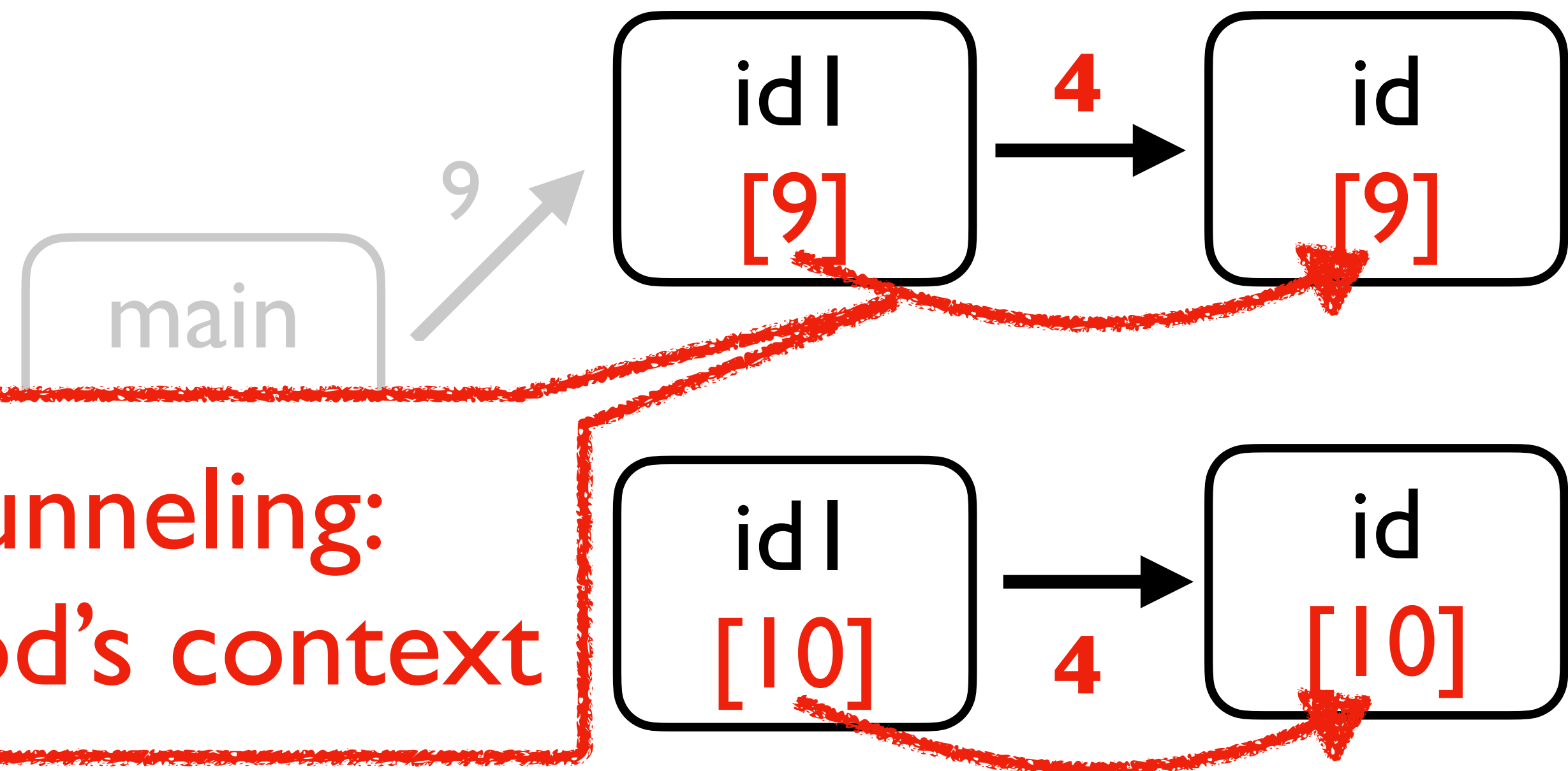
I-CFA with context tunneling  
( $T = \{4\}$ )

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

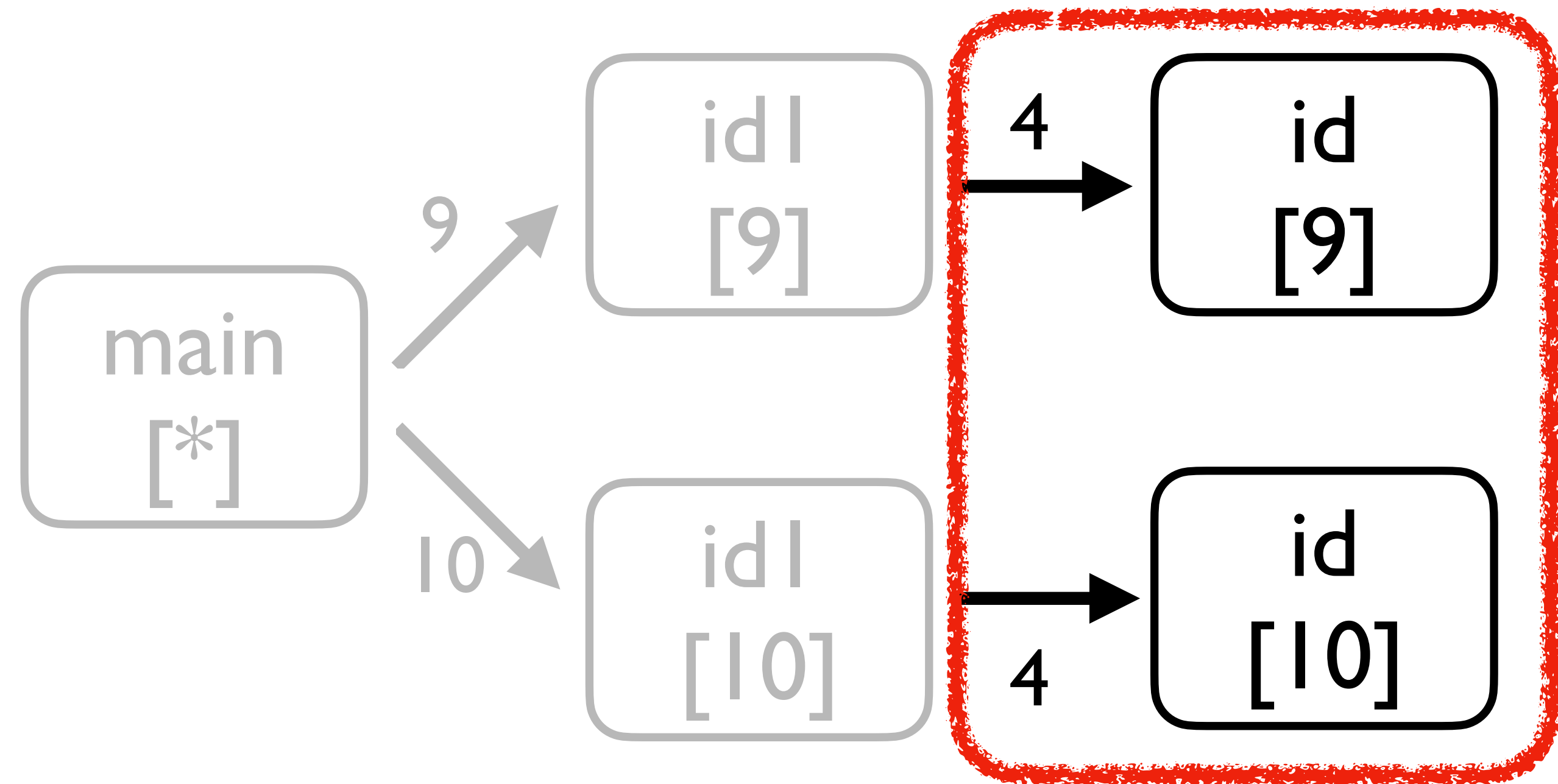


I-CFA with context tunneling  
(T = {4})

# 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:   idl(v){
4:     return id0(v);}
5: }
6: main(){
7:   c1 = new C();//C1
8:   c2 = new C();//C2
9:   a = (A) c1.idl(new A());//query1
10:  b = (B) c2.idl(new B());//query2
11: }
```



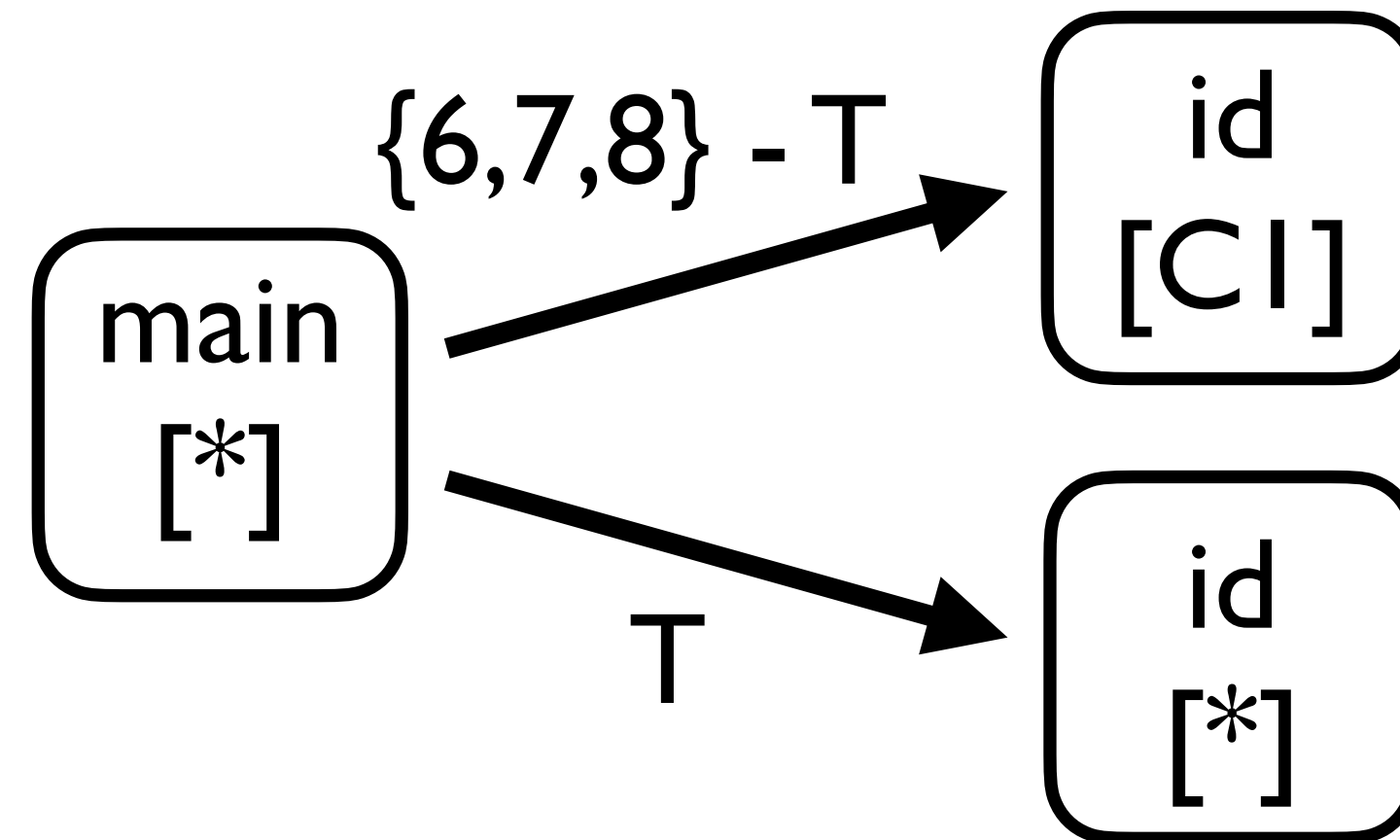
I-CFA with context tunneling  
( $T = \{4\}$ )

**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:   c1 = new C();//C1
6:   a = (A) c1.id(new A());
7:   b = (B) c1.id(new B());
8:   c = (B) c1.id(new C());
9: }
```



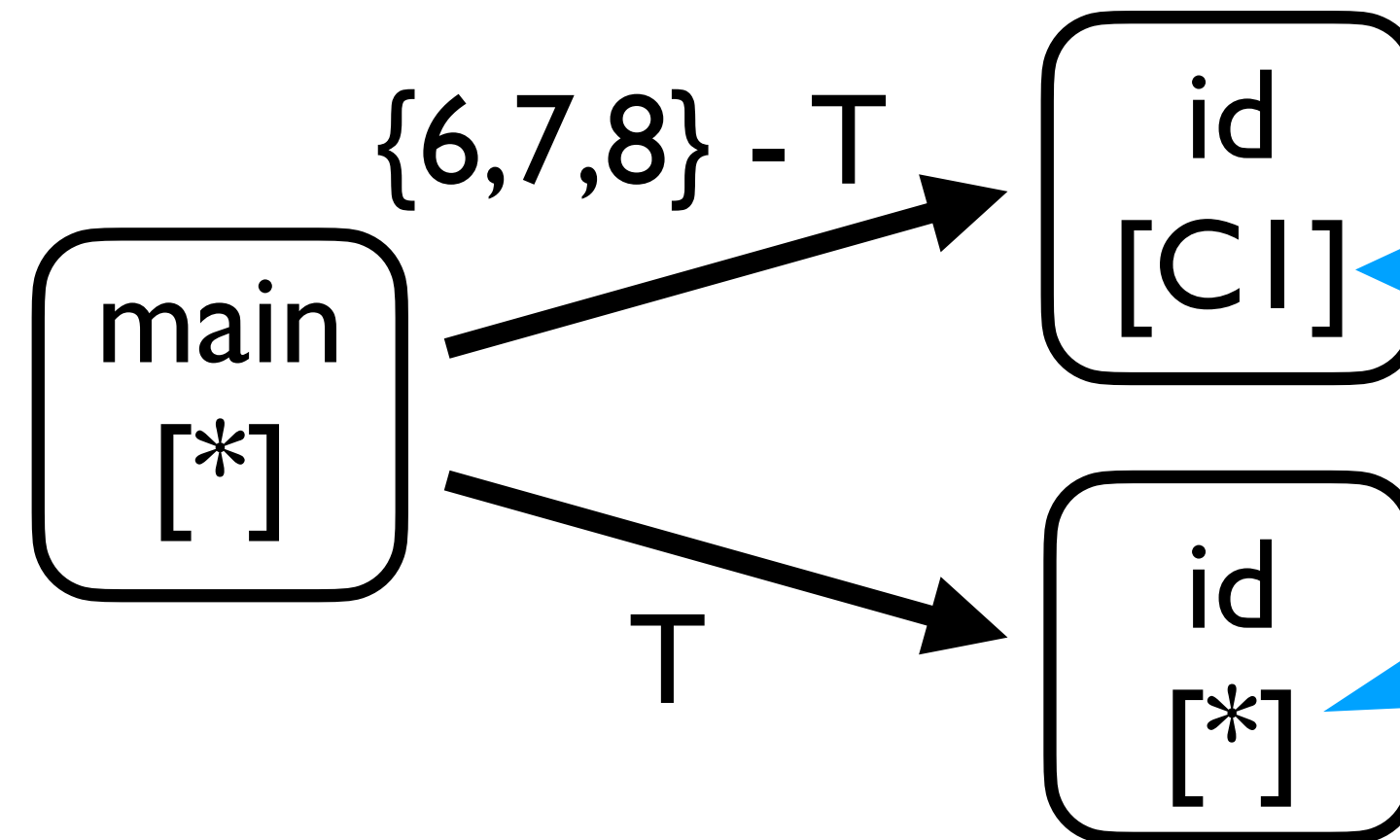
Call-graph of I-Obj with tunneling T

I-Obj + 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:   cI = new C();//CI
6:   a = (A) cI.id(new A());
7:   b = (B) cI.id(new B());
8:   c = (B) cI.id(new C());
9: }
```



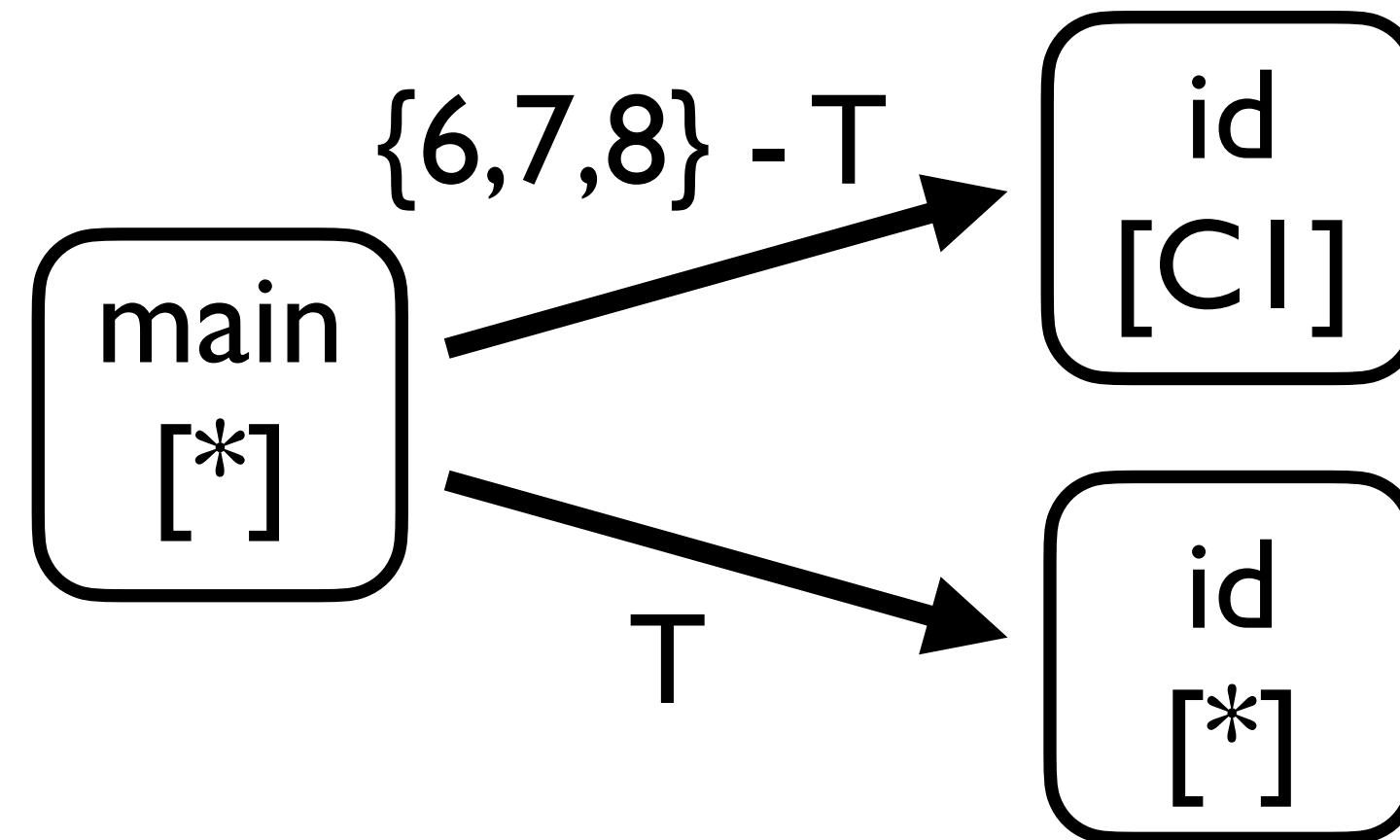
Unable to separate the three method calls with the two contexts

I-Obj + Tunneling  
(T = ?)

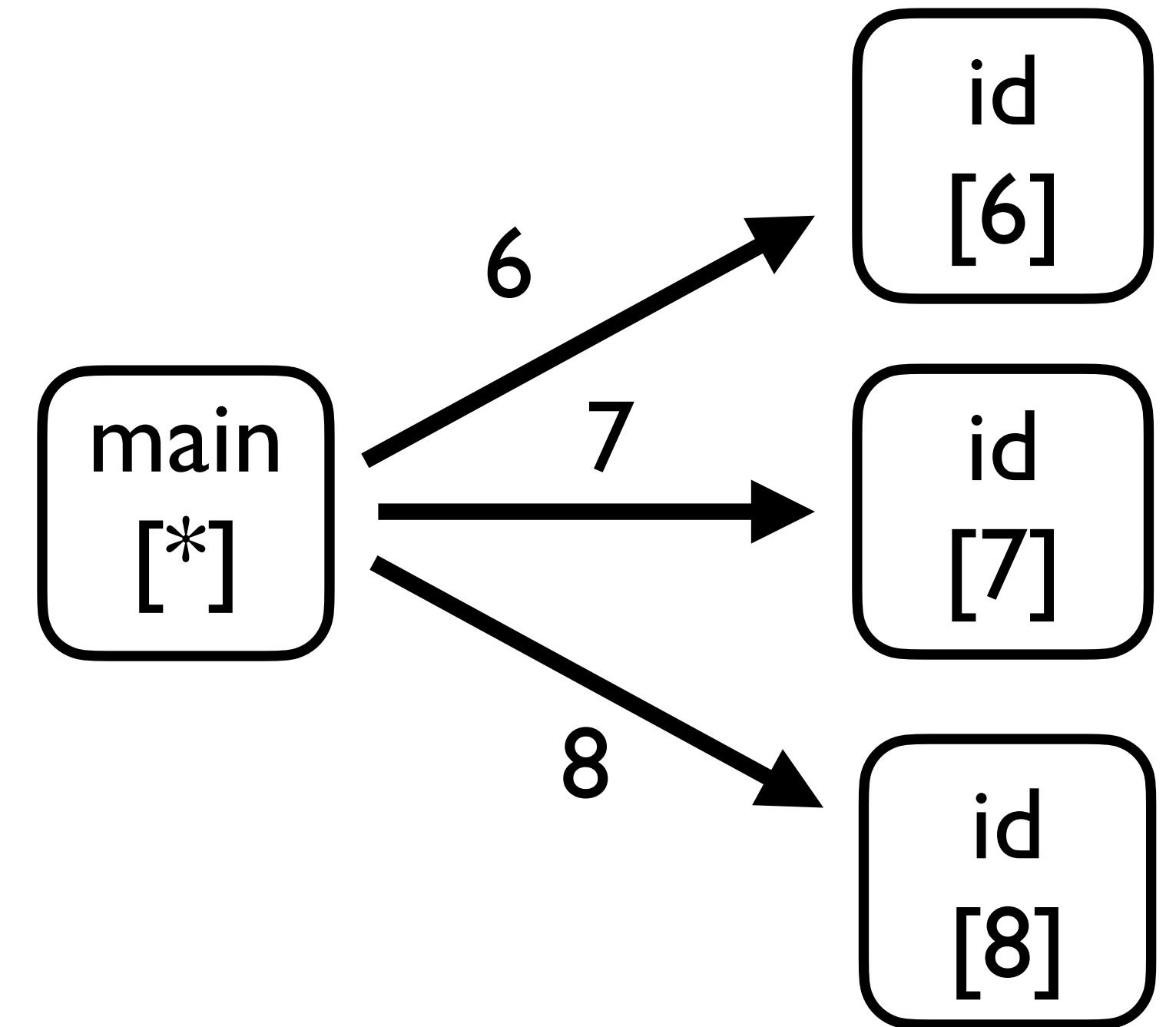
# Call-site Sensitivity vs Object Sensitivity

- Object sensitivity still suffers from its **limitation**

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



I-Obj + Tunneling  
(T = ?)



I-CFA

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

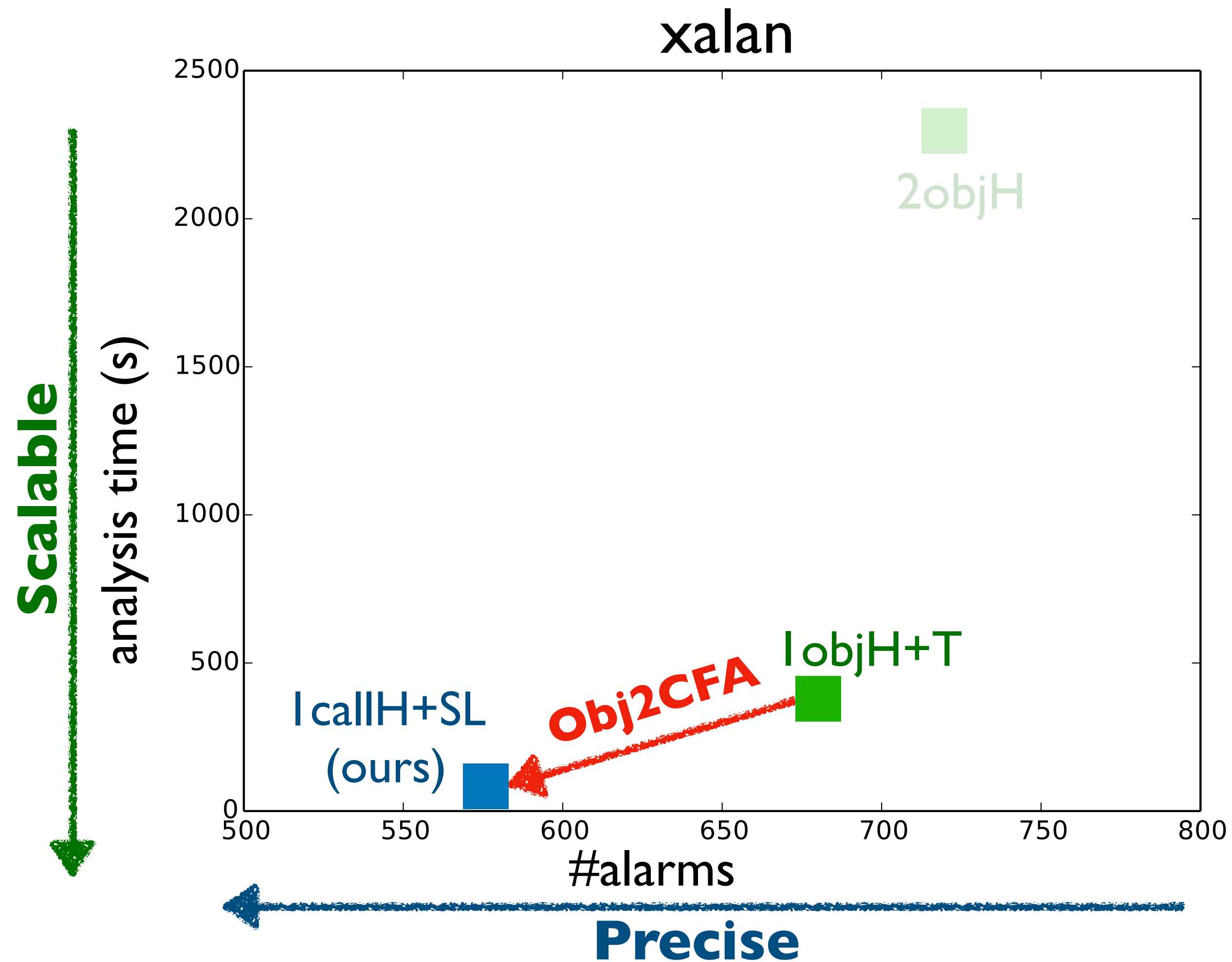
- Limitation of call-site sensitivity is **removed**
- Limitation of object sensitivity is **not removed**

## Our claim

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

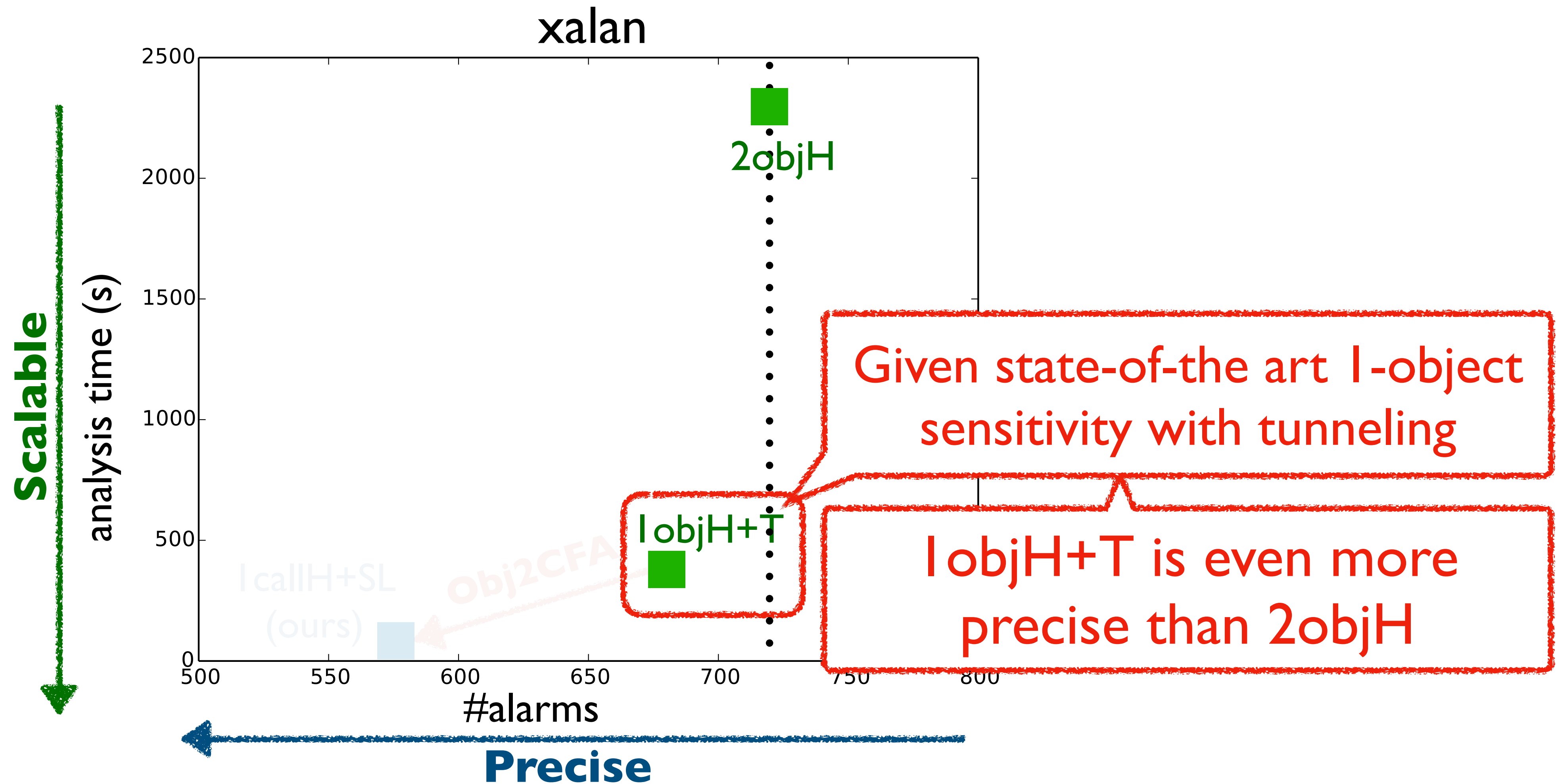
# Our Technique : **Obj2CFA**

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



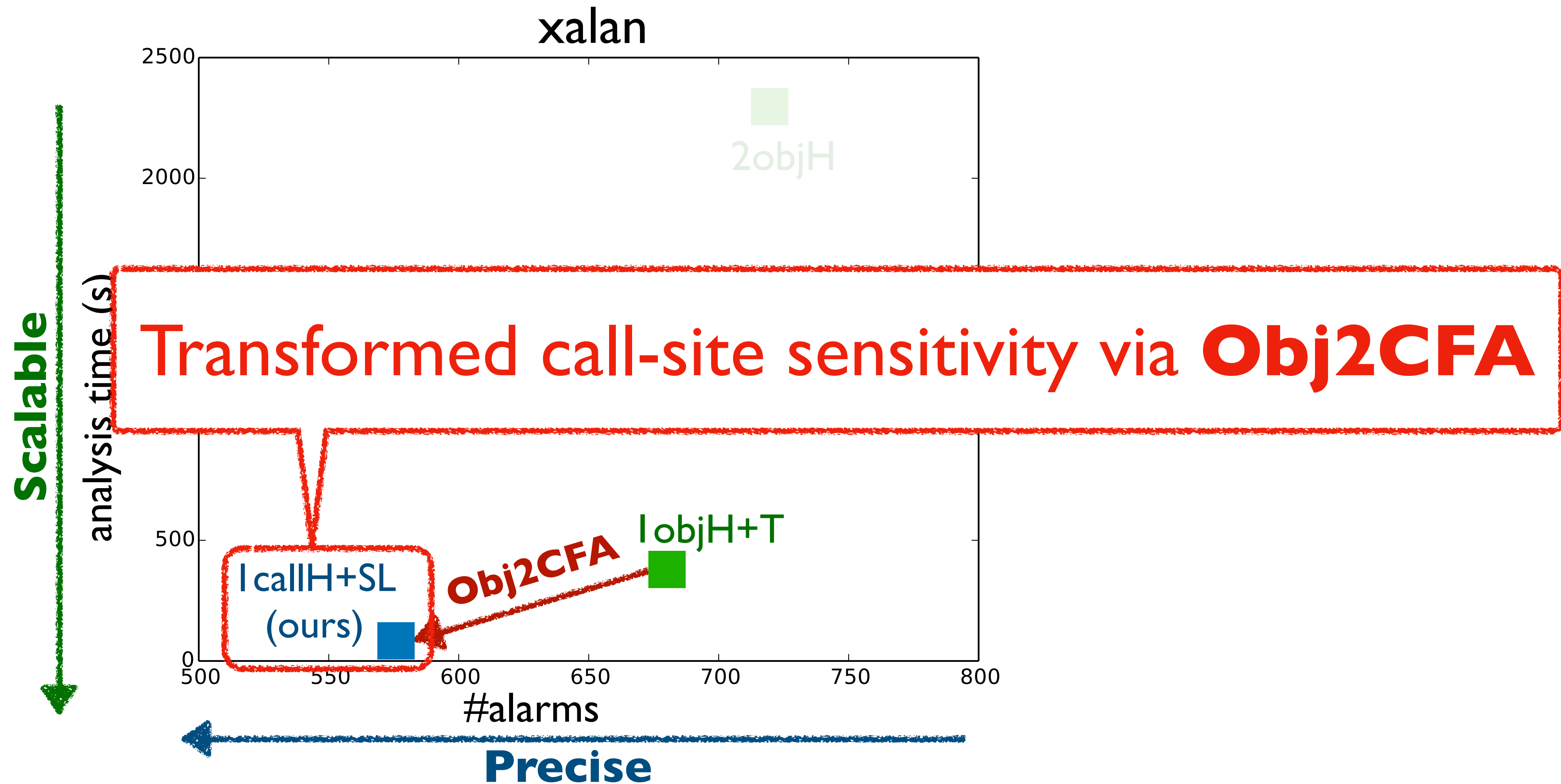
# Our Technique : **Obj2CFA**

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



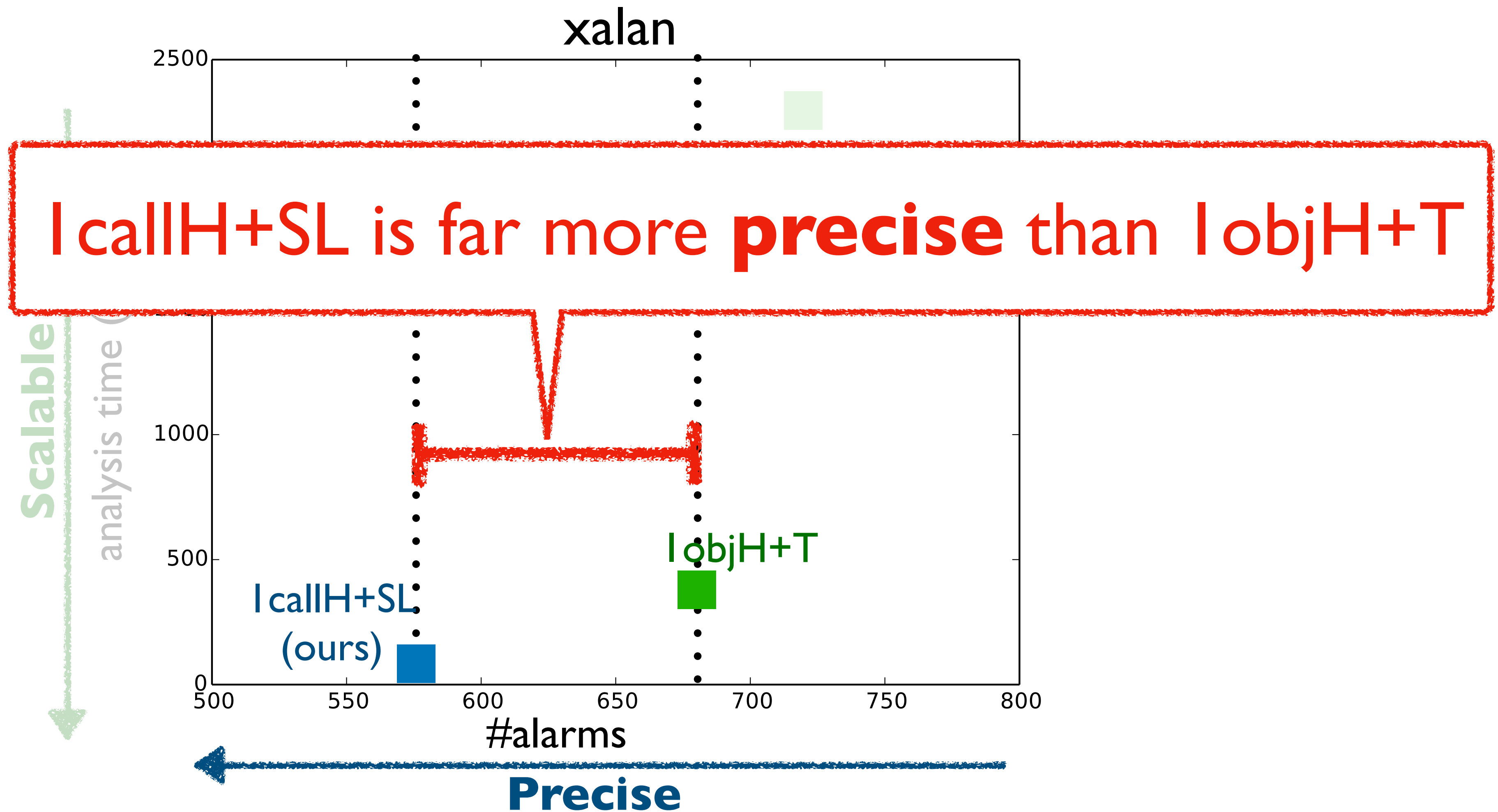
# Our Technique : **Obj2CFA**

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



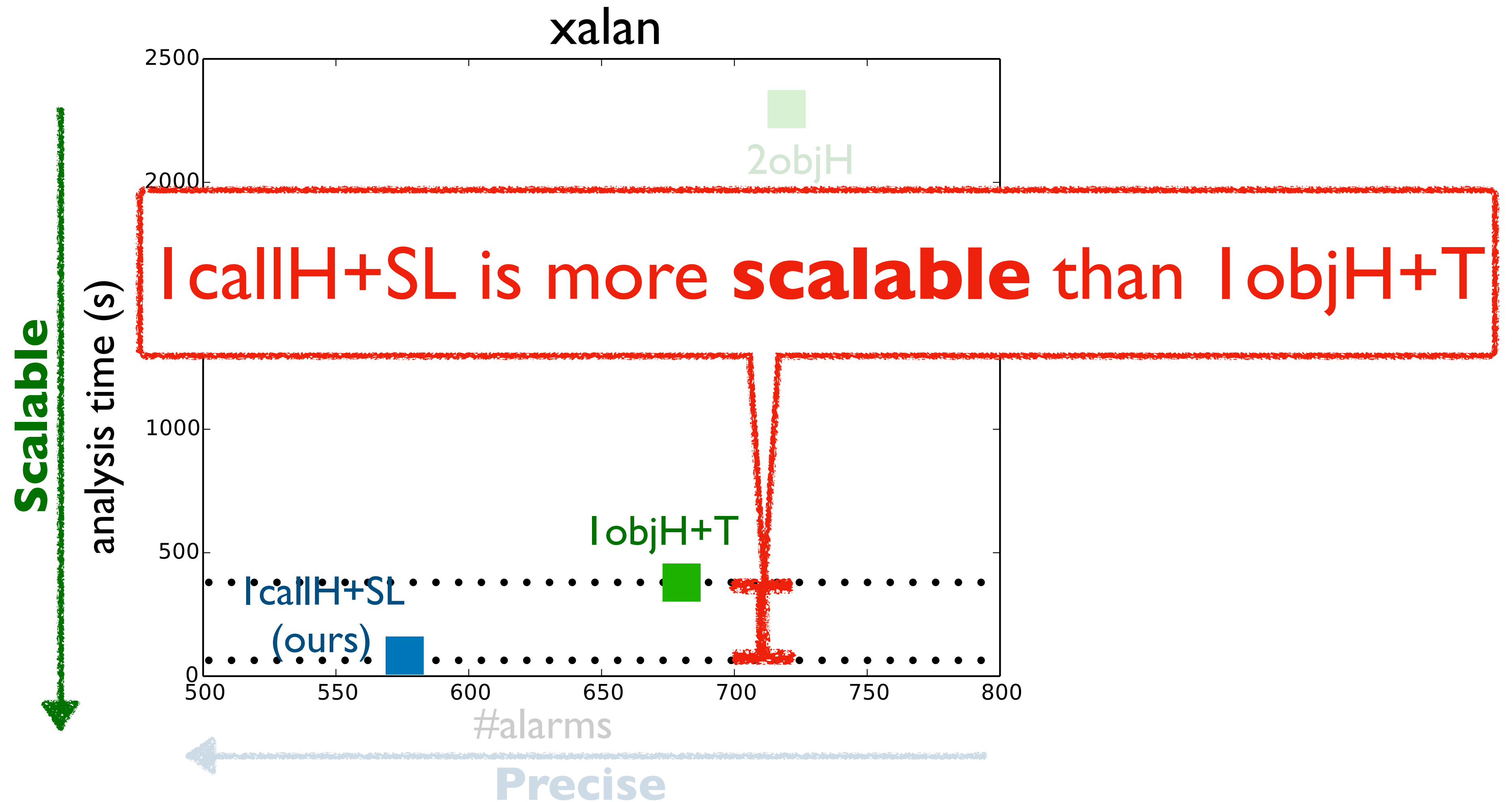
# Our Technique : **Obj2CFA**

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



# Our Technique : **Obj2CFA**

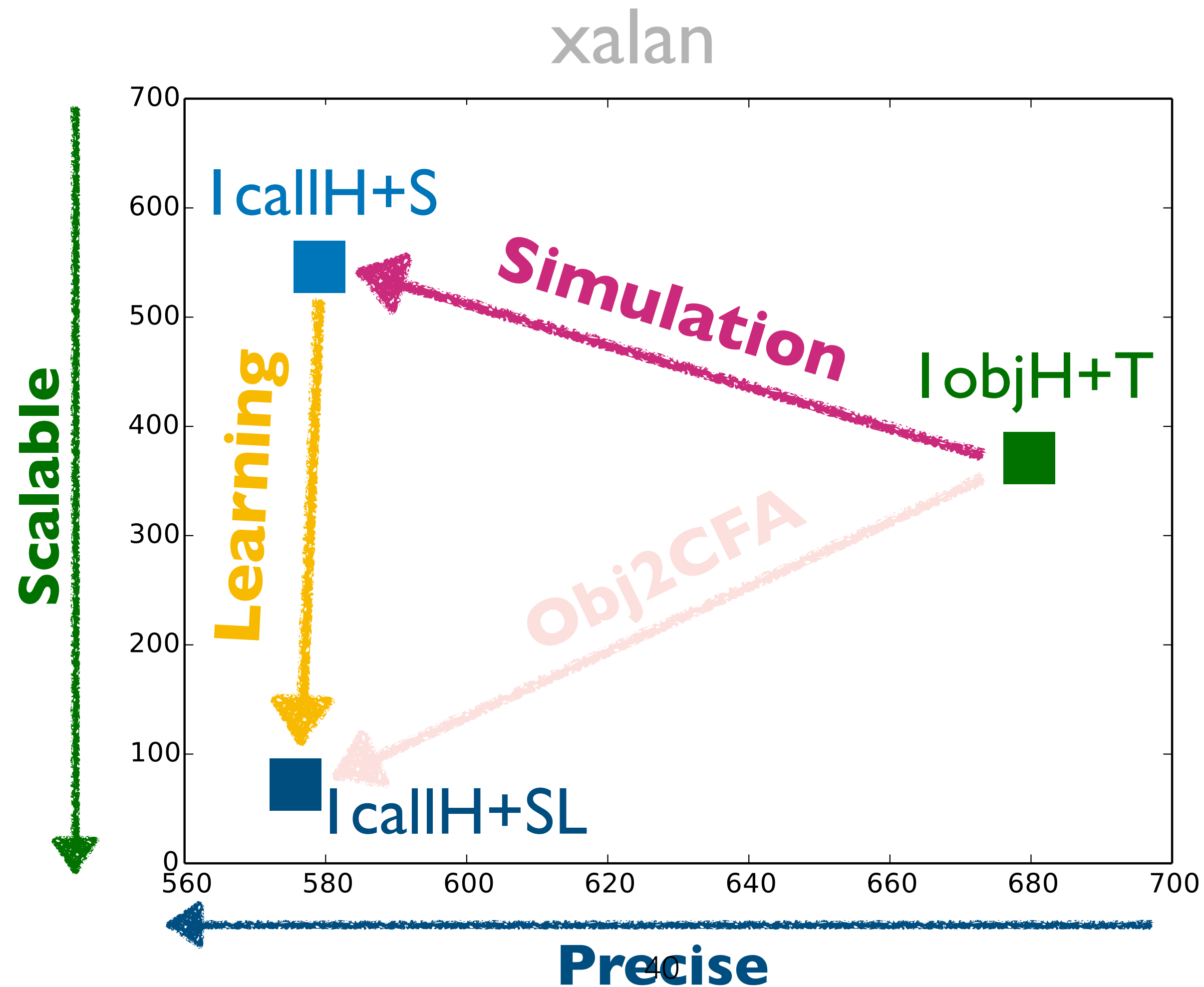
- **Obj2CFA** transforms a given **object sensitivity** into a more precise **CFA**



# **Detail of Obj2CFA**

# Our Technique : **Obj2CFA**

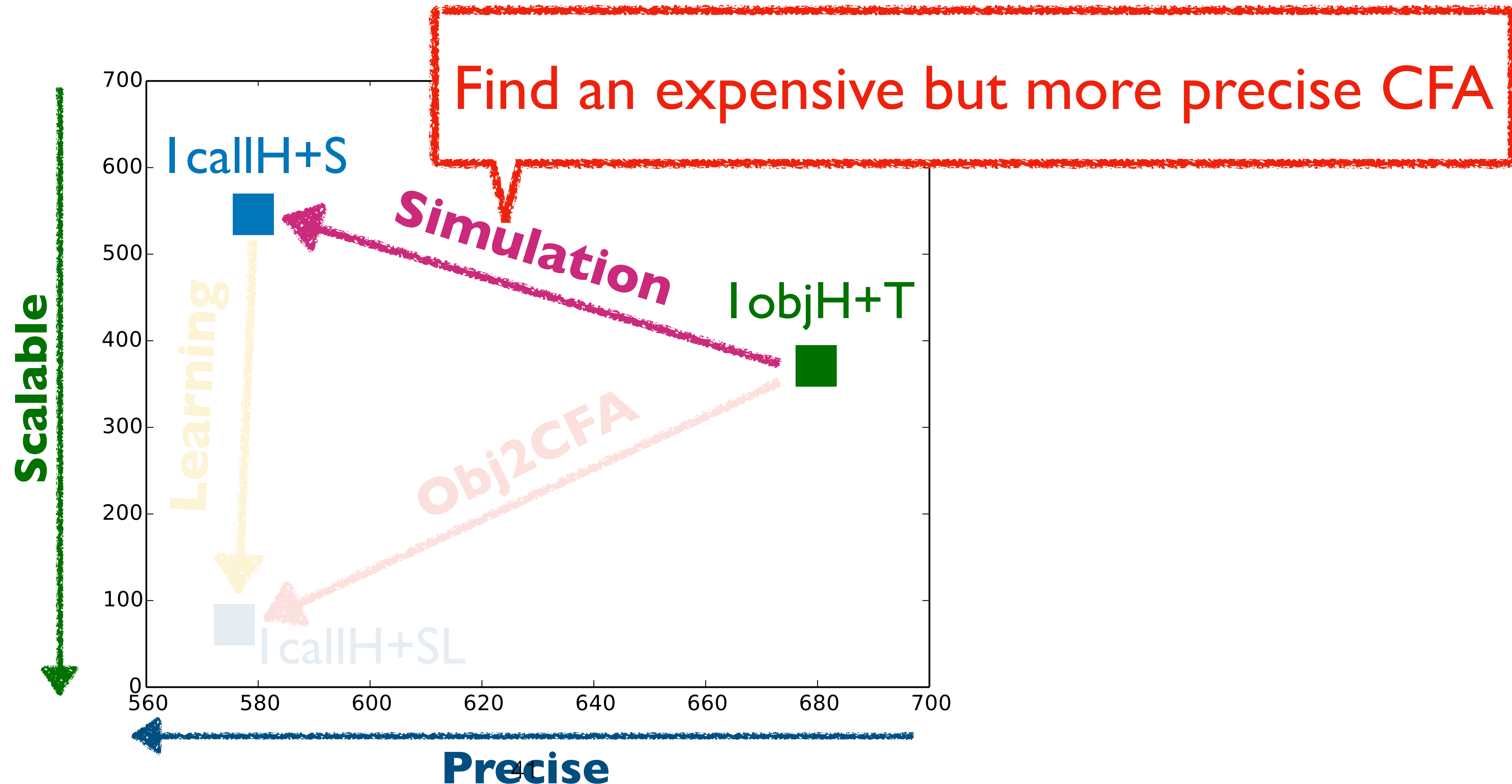
- **Obj2CFA** consists of **simulation** and simulation-guided **learning**





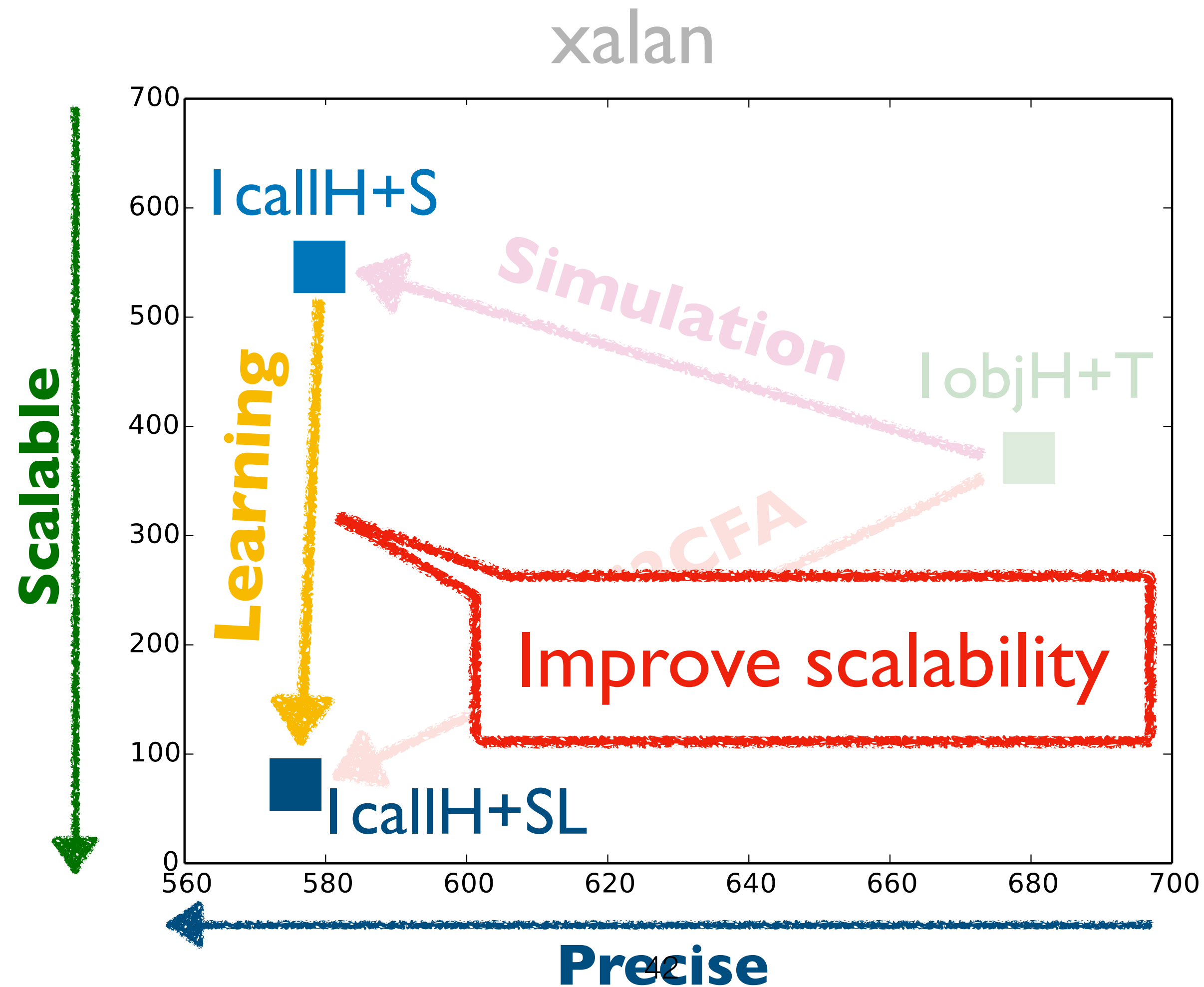
# Our Technique : **Obj2CFA**

- **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

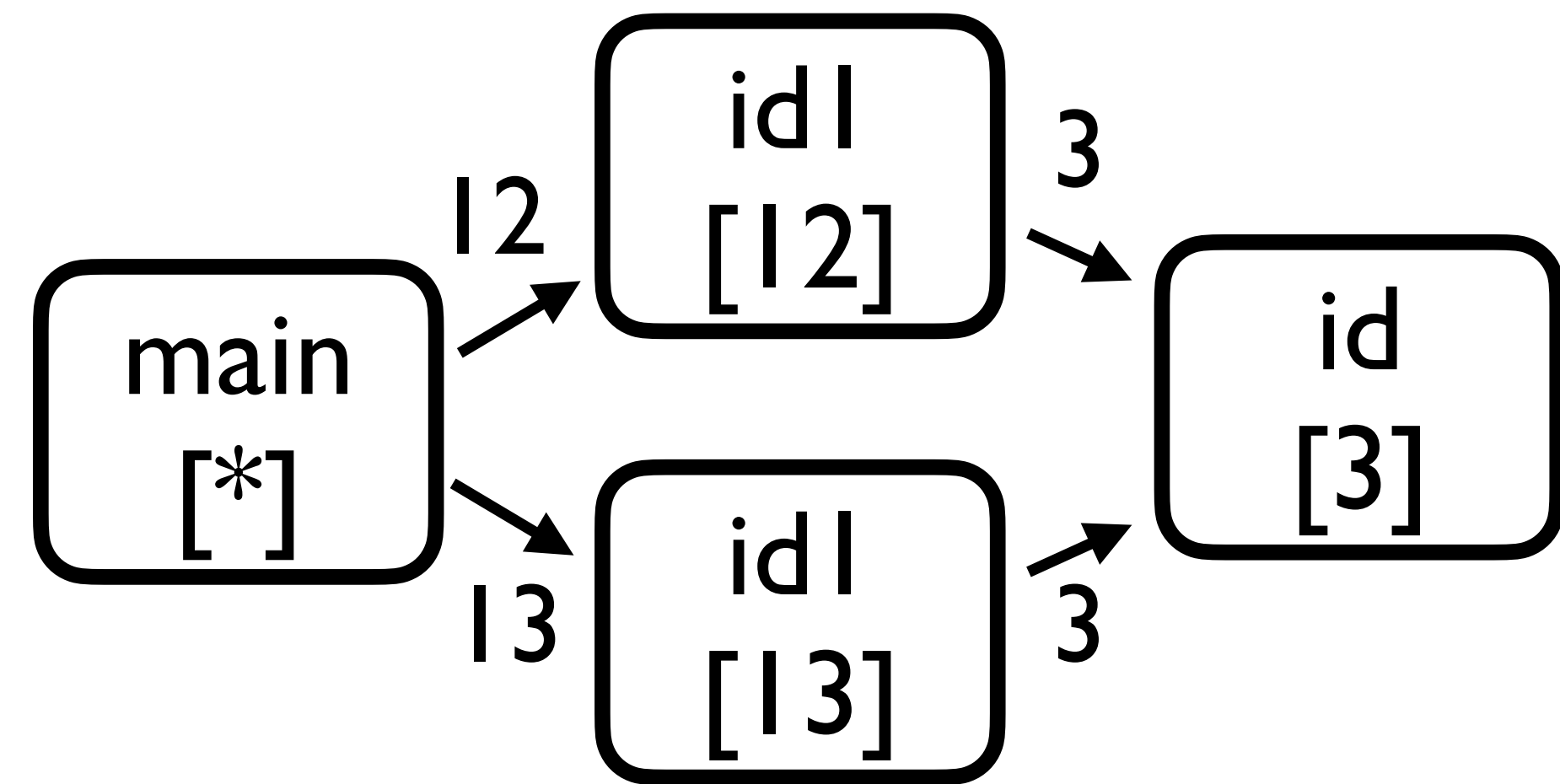
```
1: class C{
2:   id(v){return v;}
3:   idl(v){return id(v);}
4:   foo(){
5:     A a = (A) this.id(new A());}//query1
6:     B b = (B) this.id(new B());}//query2
7: }
8: main(){
9:   c1 = new C(); //C1
10:  c2 = new C(); //C2
11:  c3 = new C(); //C3
12:  A a = (A) c1.idl(new A()); //query3
13:  B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

# Technique I: Simulation

- Running example to illustrate Simulation

```
1: class C{
2:   id(v){return v;}
3:   idl(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.idl(new A()); //query3
13:  B b = (B) c2.idl(new B()); //query4
14:  c3.foo();
15: }
```

Limitation of conventional I-CFA



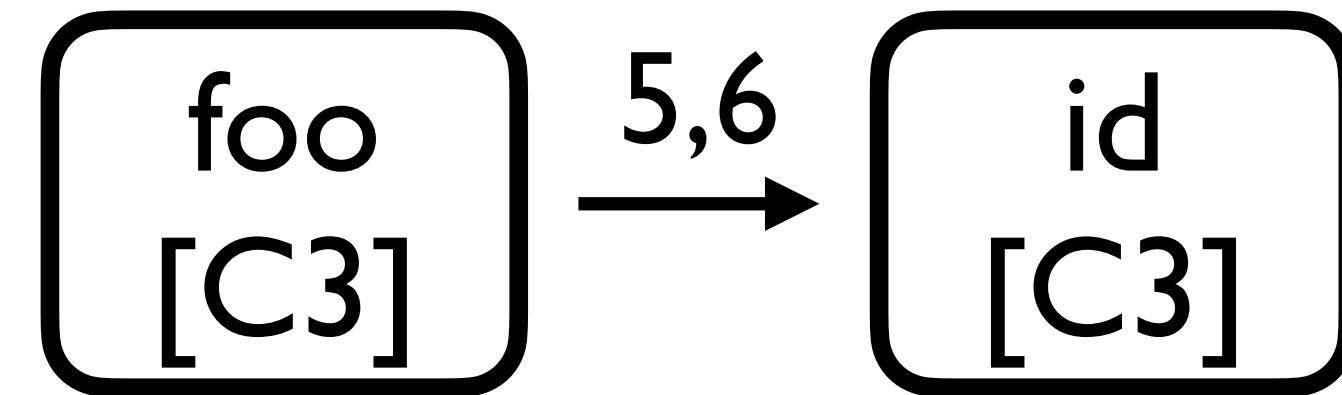
# Technique I: Simulation

- Running example to illustrate Simulation

```
1: class C{
2:   id(v){return v;}
3:   idl(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.idl(new A()); //query3
13:  B b = (B) c2.idl(new B()); //query4
14:  c3.foo();
15: }
```



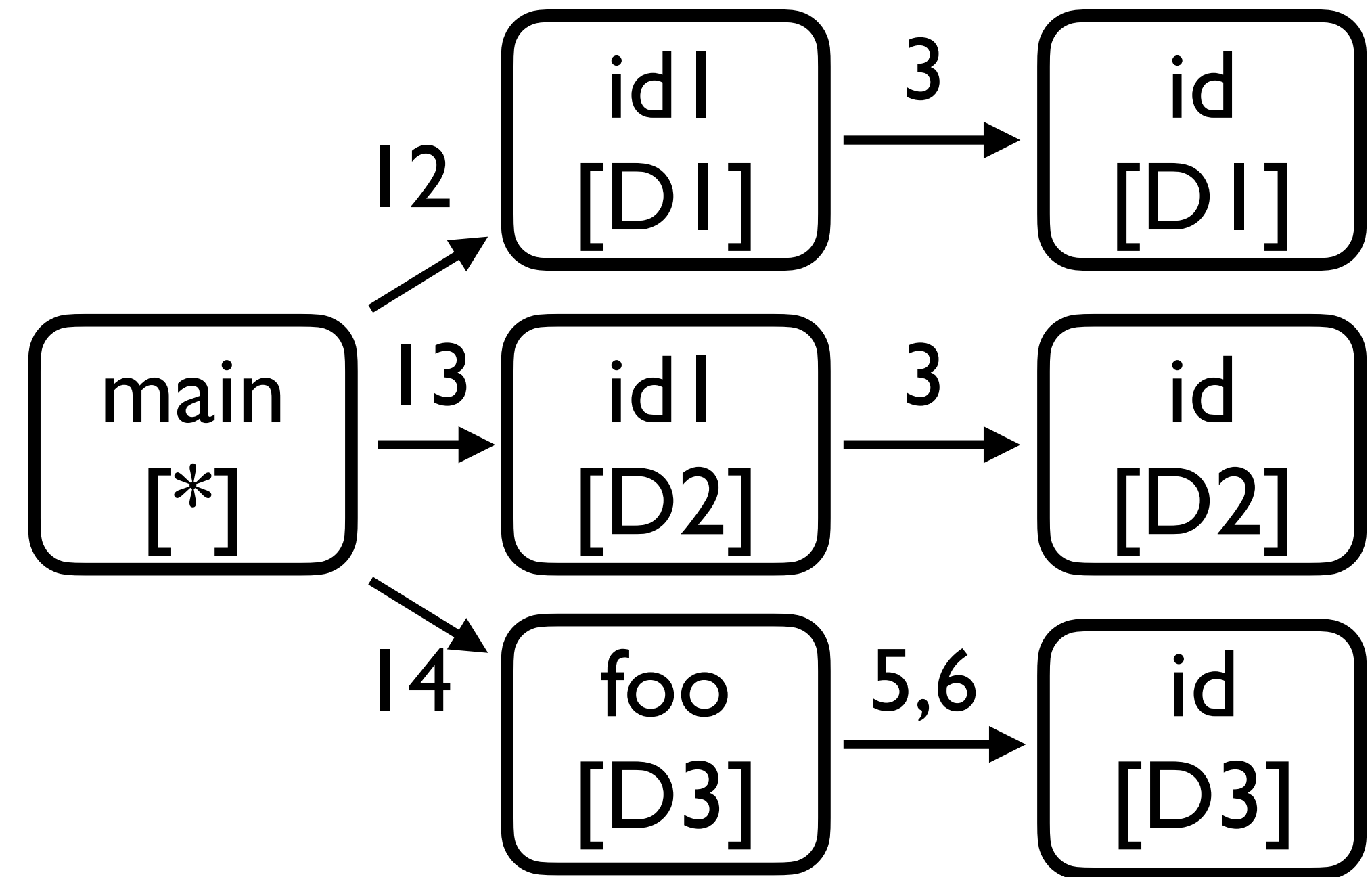
Limitation of object sensitivity



# Technique 1: Simulation

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

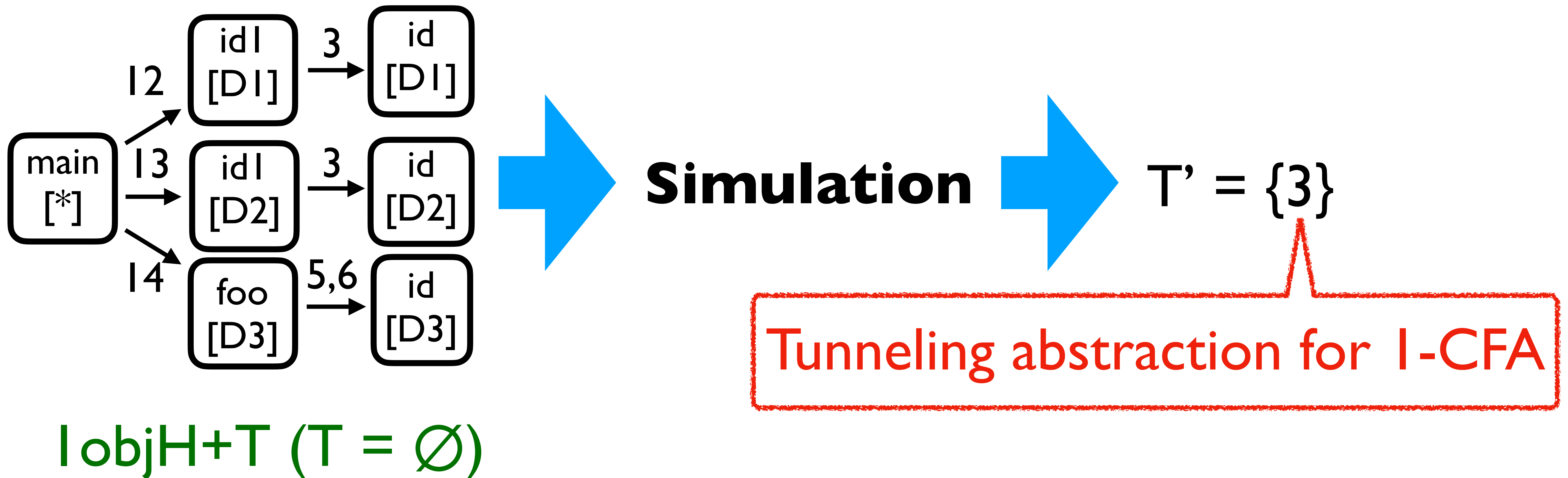
```
1: class C{
2:   id(v){return v;}
3:   idl(v){return id(v);}
4:   foo(){
5:     A a = (A) this.id(new A()); //query1
6:     B b = (B) this.id(new B()); //query2
7:   }
8: main(){
9:   c1 = new C(); //C1
10:  c2 = new C(); //C2
11:  c3 = new C(); //C3
12:  A a = (A) c1.idl(new A()); //query3
13: 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



**lobjH+T** ( $T = \emptyset$ )

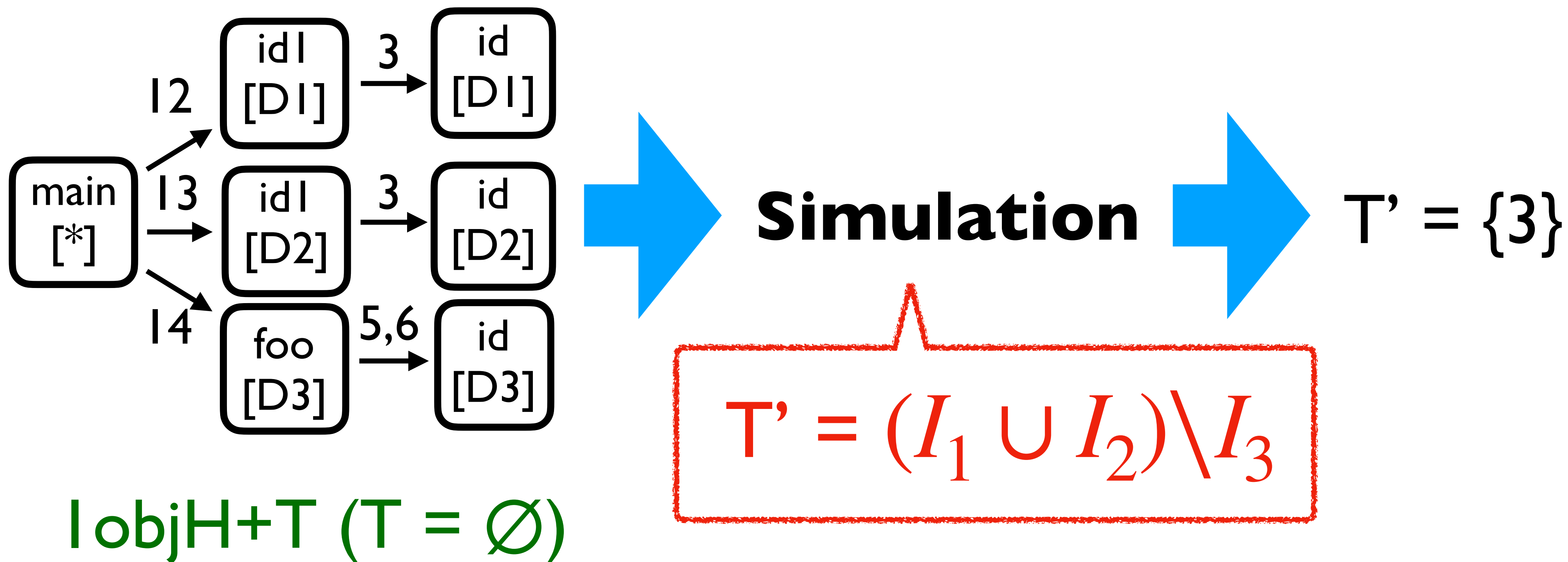
# Technique 1: Simulation

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



# Technique 1: Simulation

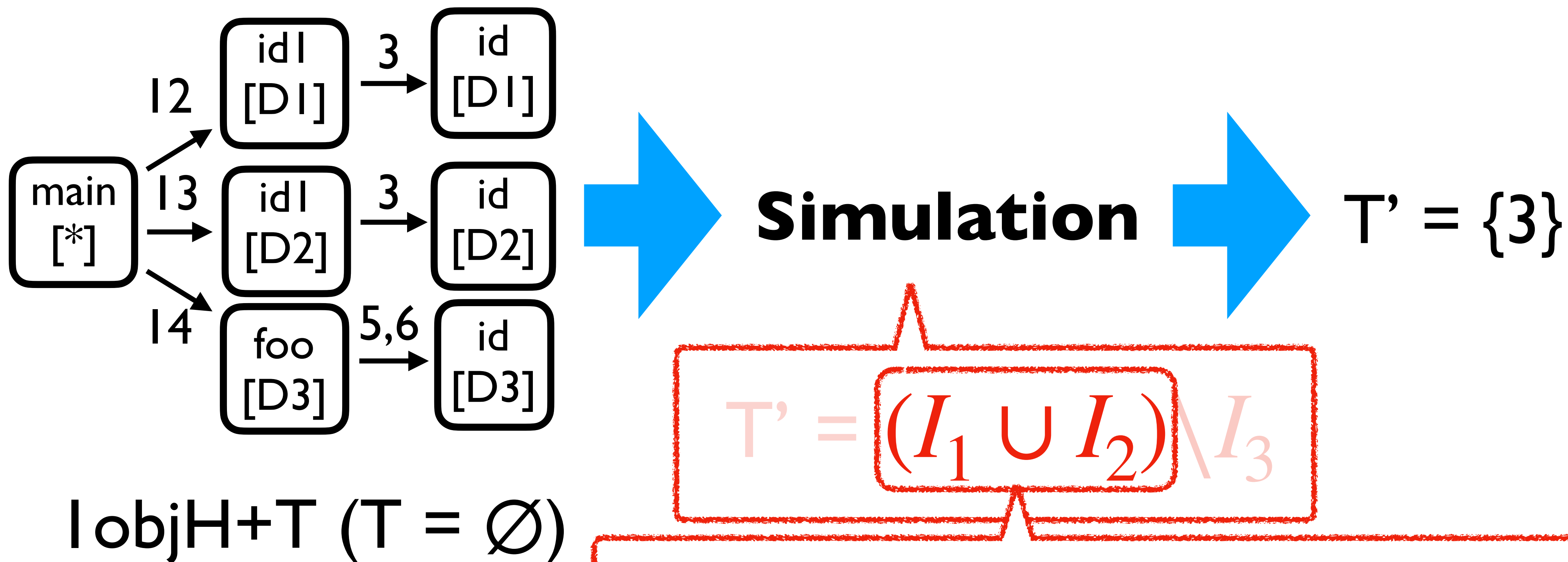
- **Simulation** takes a call-graph and produce a tunneling abstraction for CFA





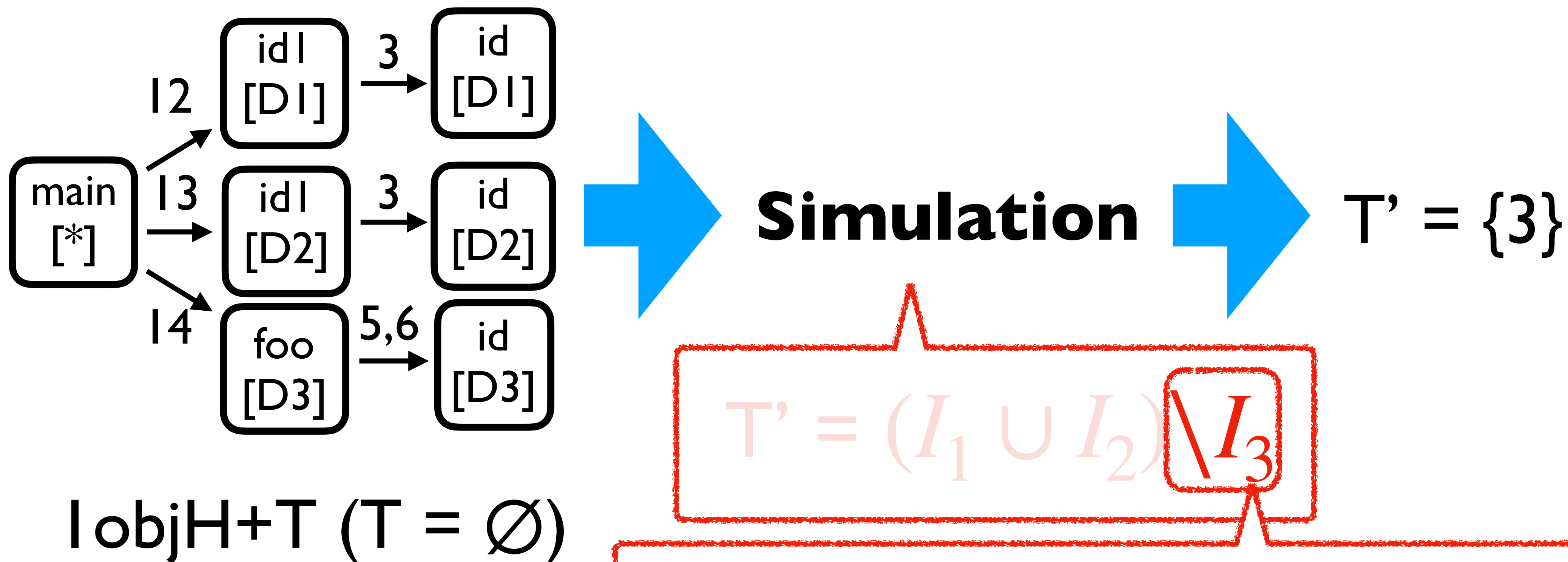
# Technique 1: Simulation

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



# Technique 1: Simulation

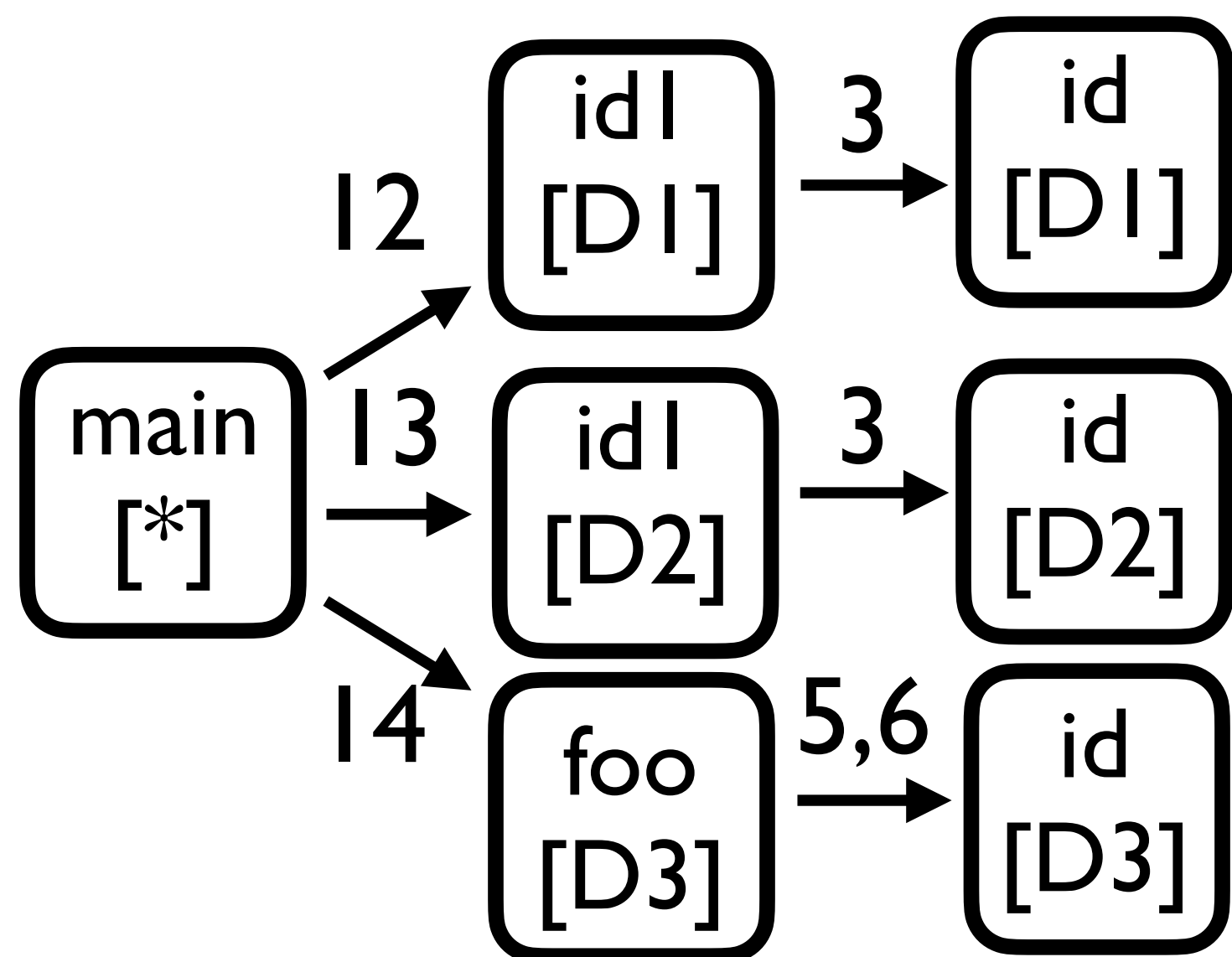
- **Simulation** takes a call-graph and produce a tunneling abstraction for CFA



Tunneling should be avoided for improving precision

# Technique 1: Simulation

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



## Intuition of Simulation

Suppose the call-graph is produced from 1-CFA + T' and infer the T'

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

$I_{call}H+T'$

What is T'?

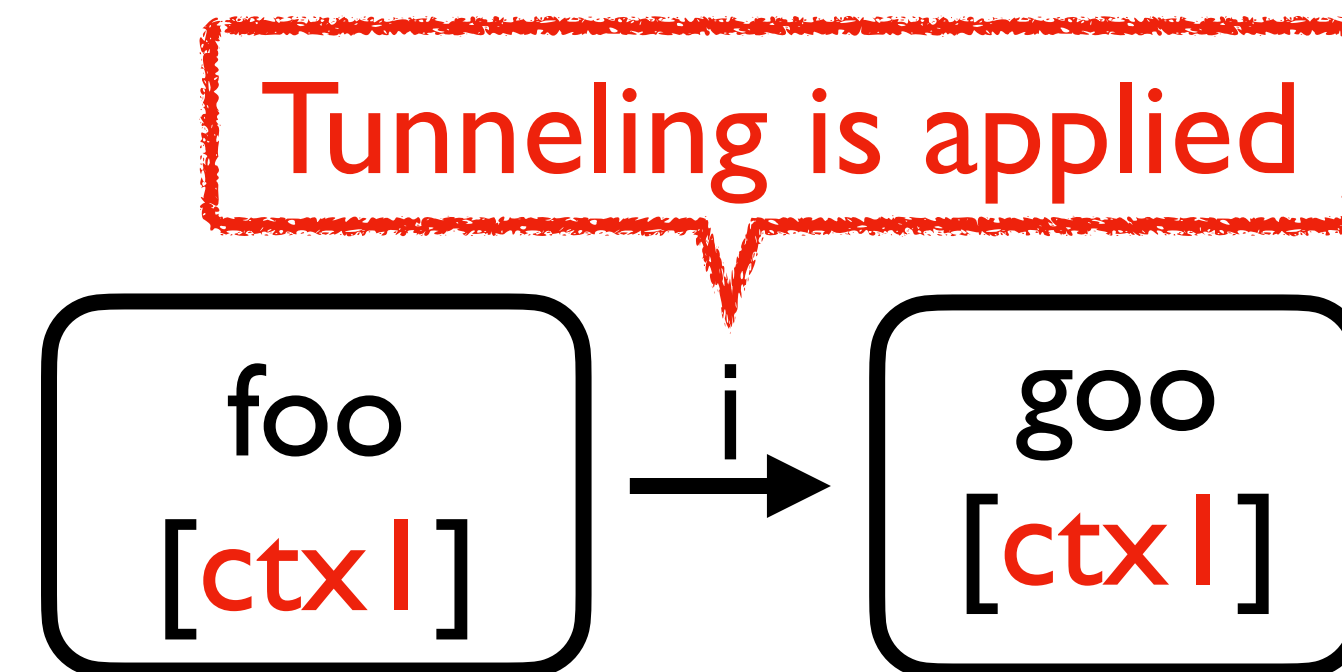
# Intuition Behind Simulation ( $I_1 \cup I_2$ )

- 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

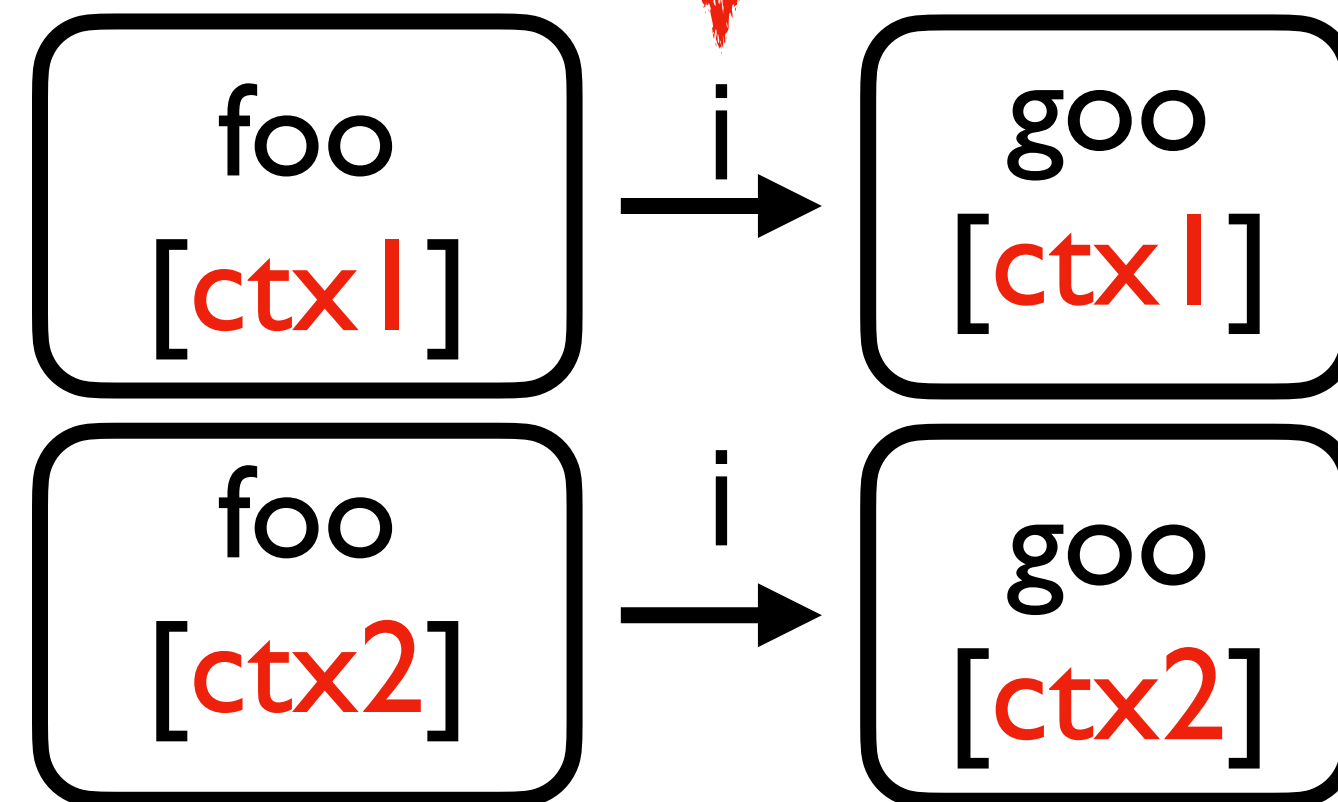
$I_1$

- Property 1: caller and callee methods have the **same context**

# Intuition Behind Simulation ( $I_1 \cup I_2$ )

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

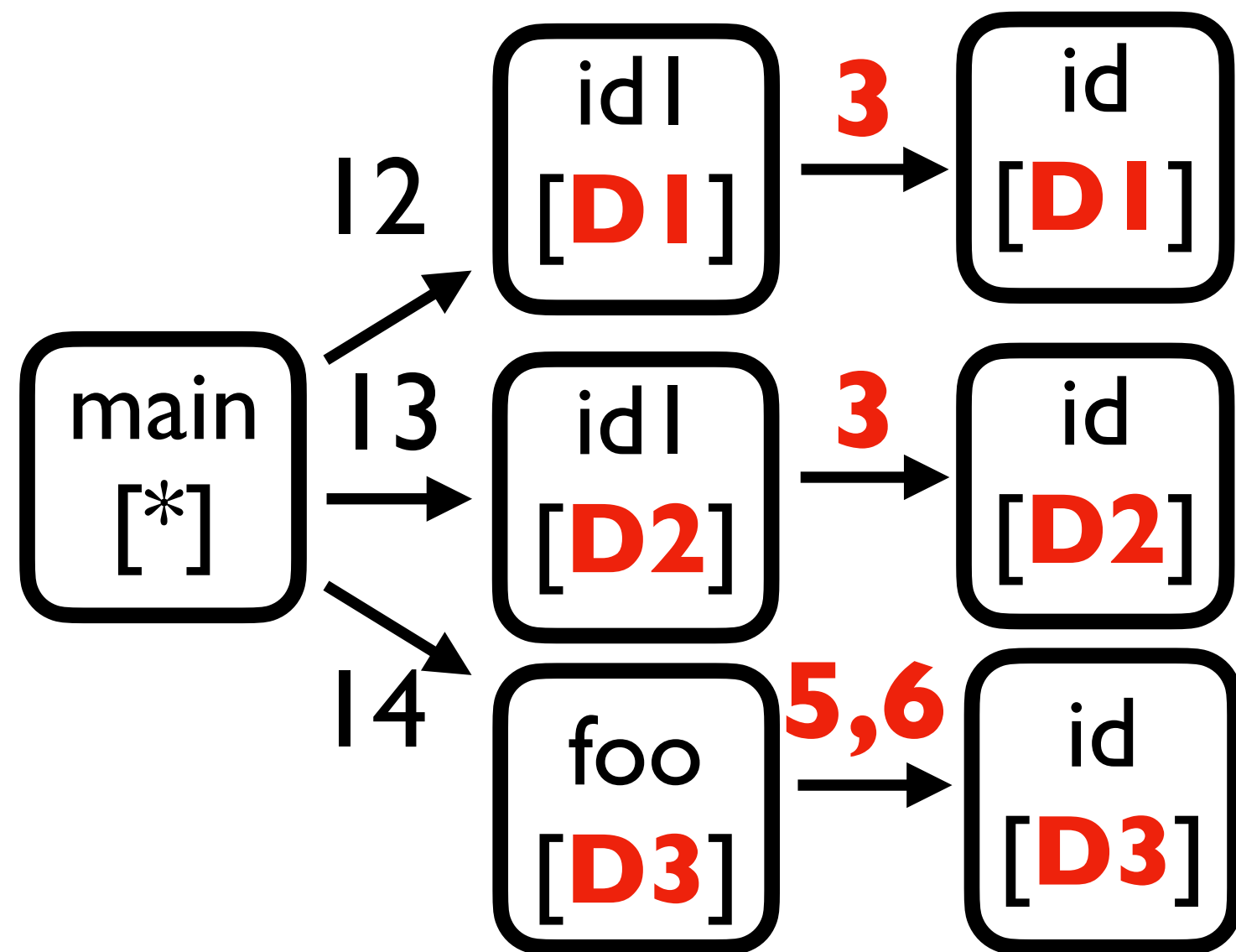
Tunneling is applied



- **Property of context tunneled invocations**
- **Property 2: different caller contexts imply different callee contexts**

# Intuition Behind Simulation ( $I_1 \cup I_2$ )

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



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

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

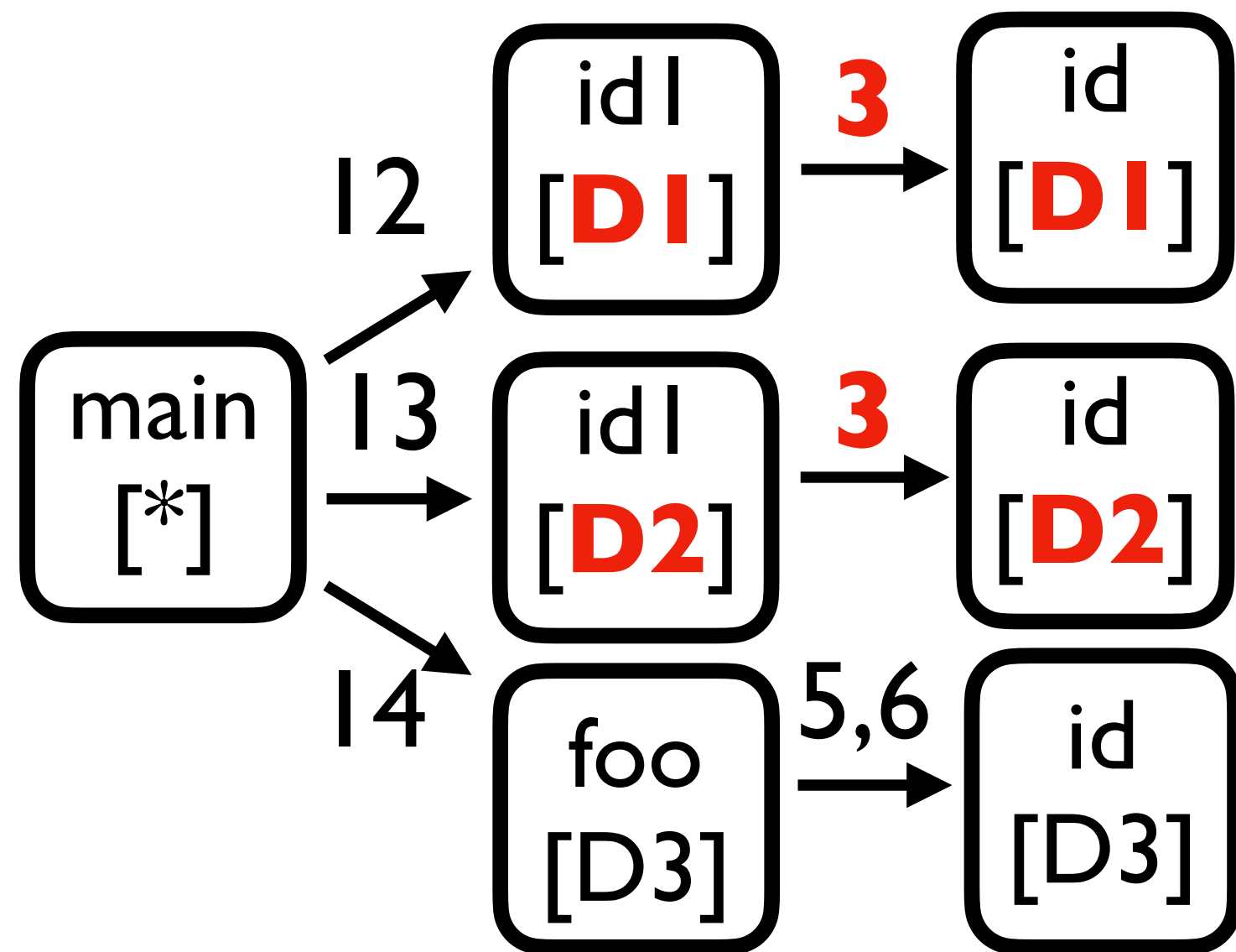
~~$I \text{ objH} + T$  ( $T = \emptyset$ )~~

$I \text{ callH} + T'$

What is  $T'$ ?

# Intuition Behind Simulation ( $I_1 \cup I_2$ )

- Suppose given call-graph is produced from  $I_{callH+T}$  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

$$I_2 = \{3\}$$

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

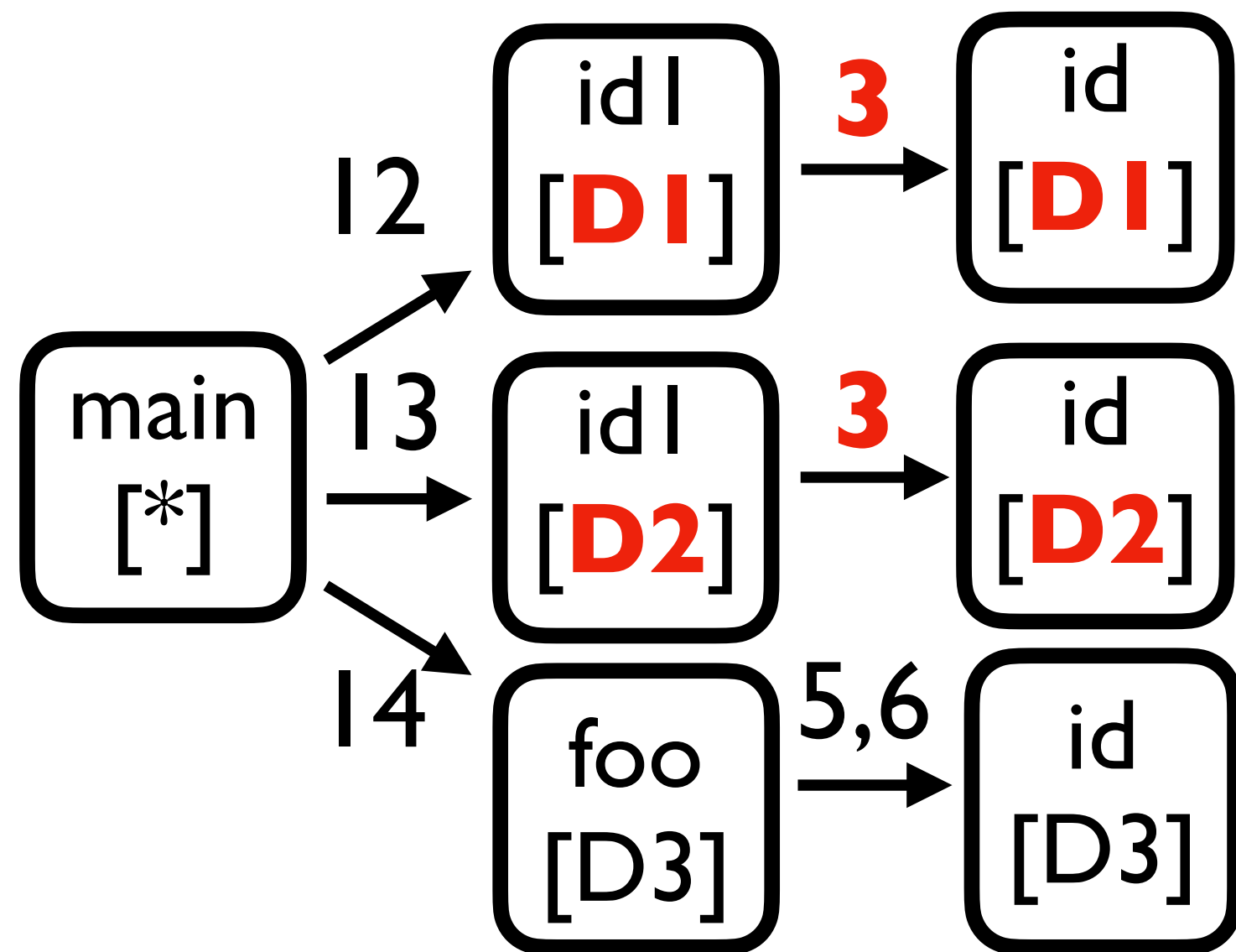
$I_{callH+T}$

What is  $T$ ?



# Intuition Behind Simulation ( $I_1 \cup I_2$ )

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



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

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

- $I_2$ : different caller ctx imply different callee ctx

$$I_2 = \{3\}$$

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

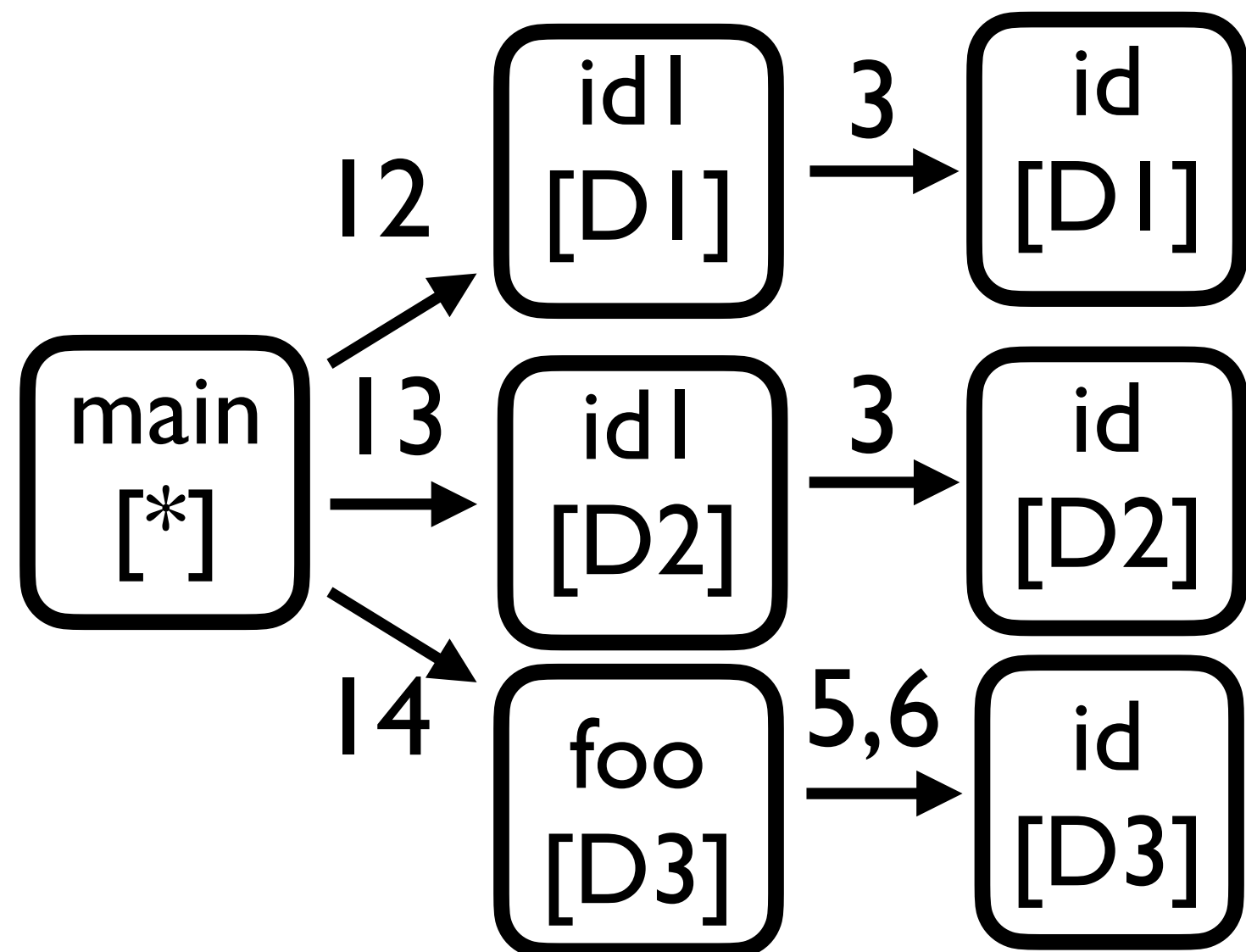
~~$I \text{ objH} + T$  ( $T = \emptyset$ )~~

$I \text{ callH} + T'$

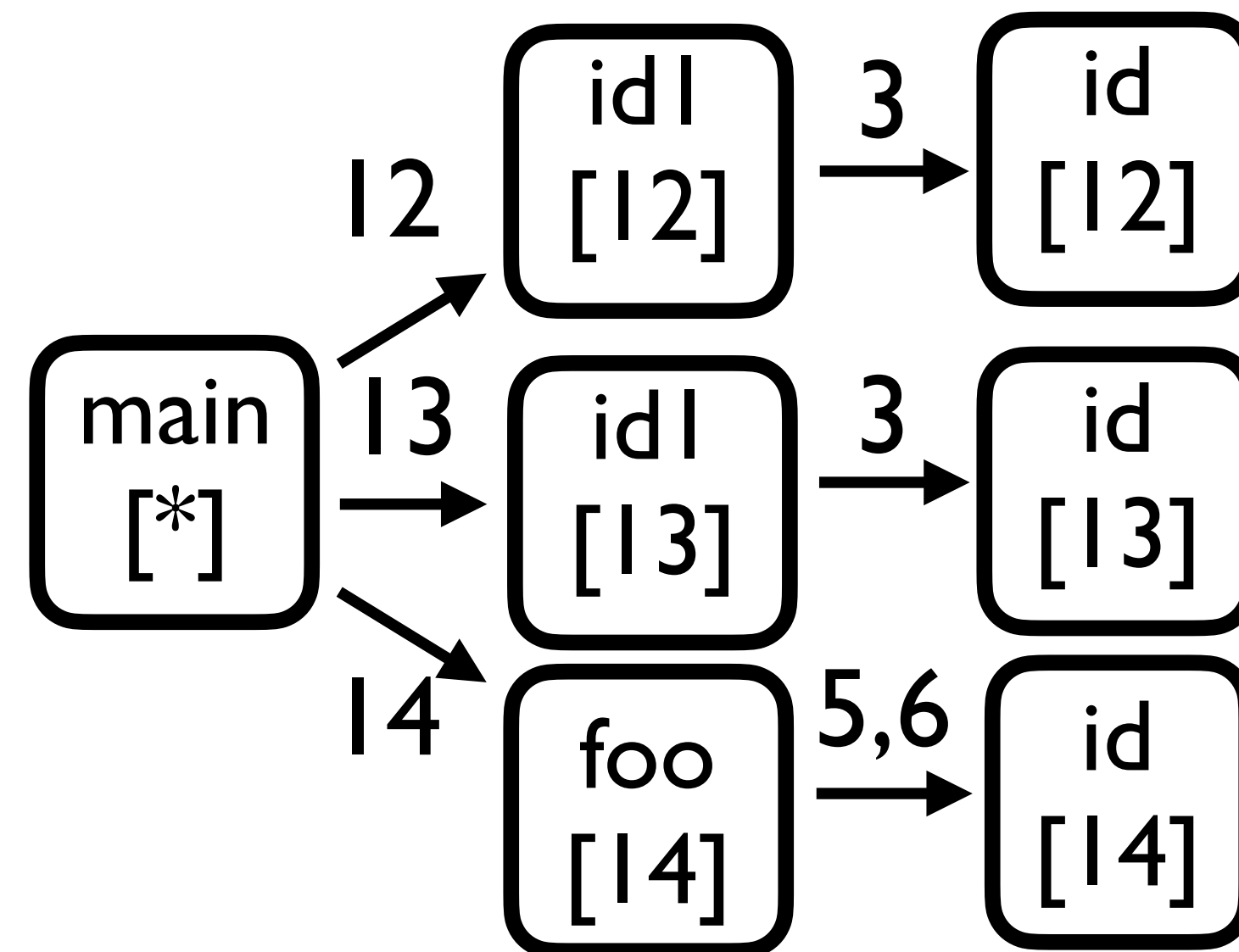
What is  $T'$ ?

# Intuition Behind Simulation ( $I_1 \cup I_2$ )

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



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

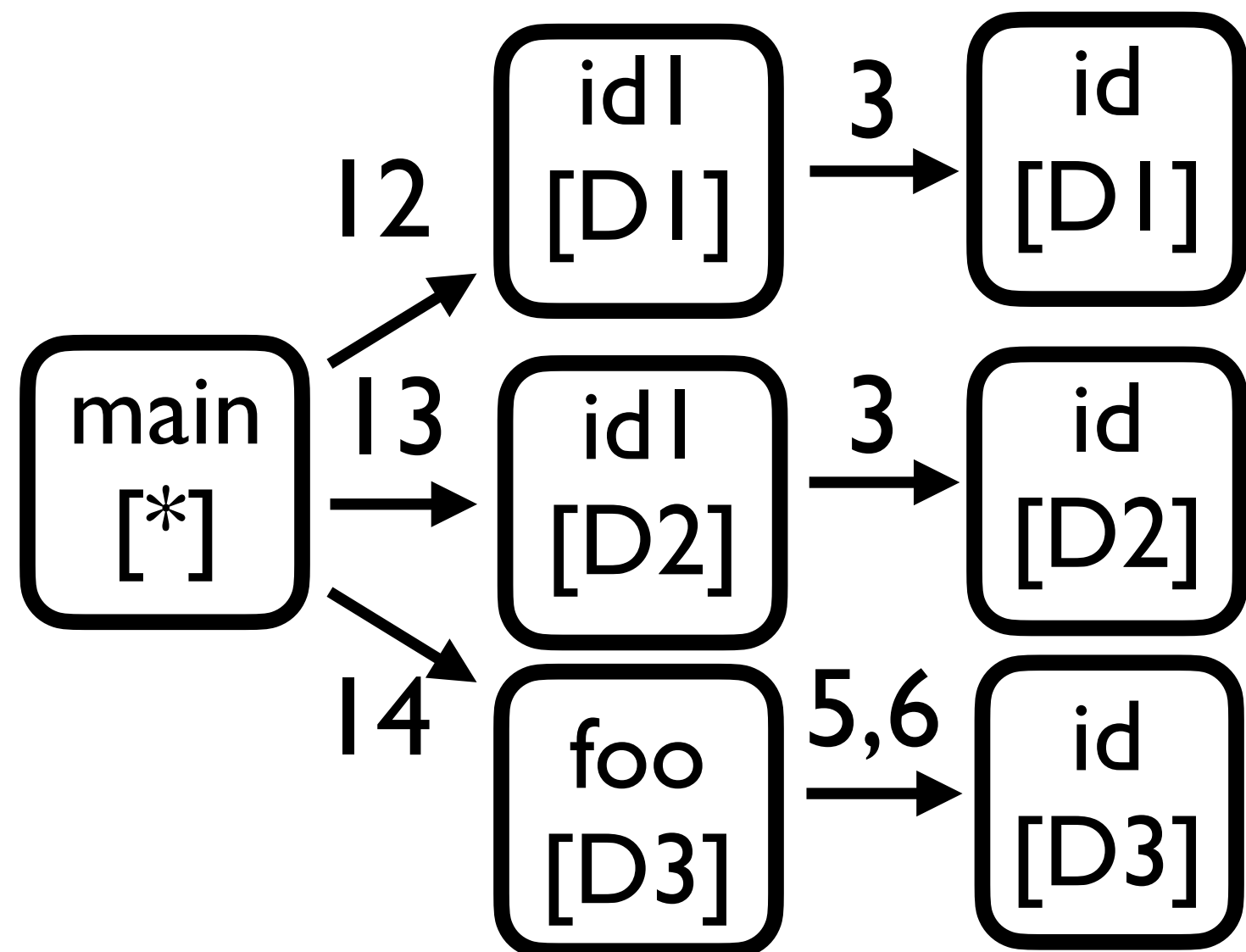


$I \text{ callH}+T'$  ( $T' = \{3,5,6\}$ )

# Intuition Behind Simulation ( $I_1 \cup I_2$ )

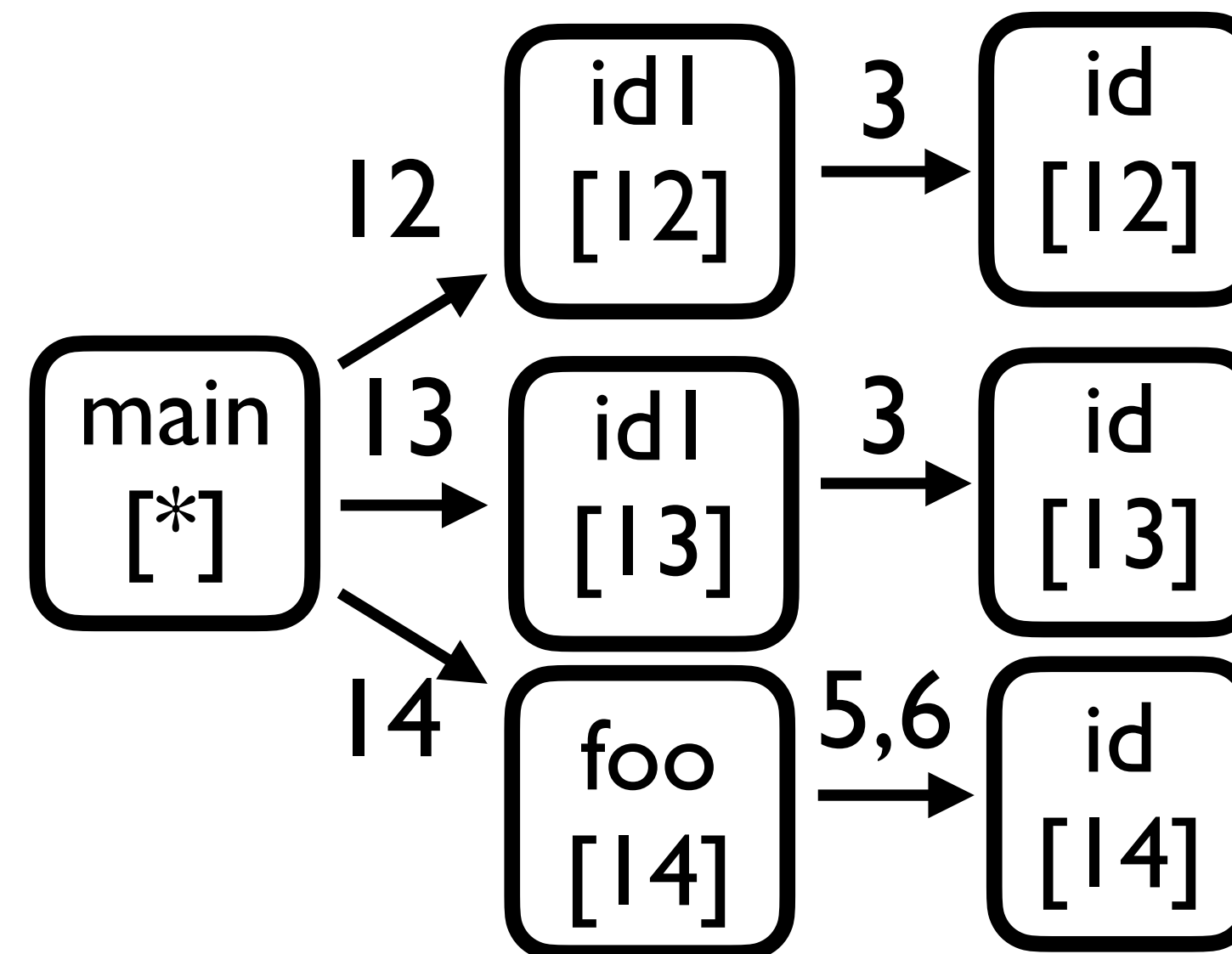
- Suppose given call-graph and infer what  $T'$  is

Exactly the same analyses



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

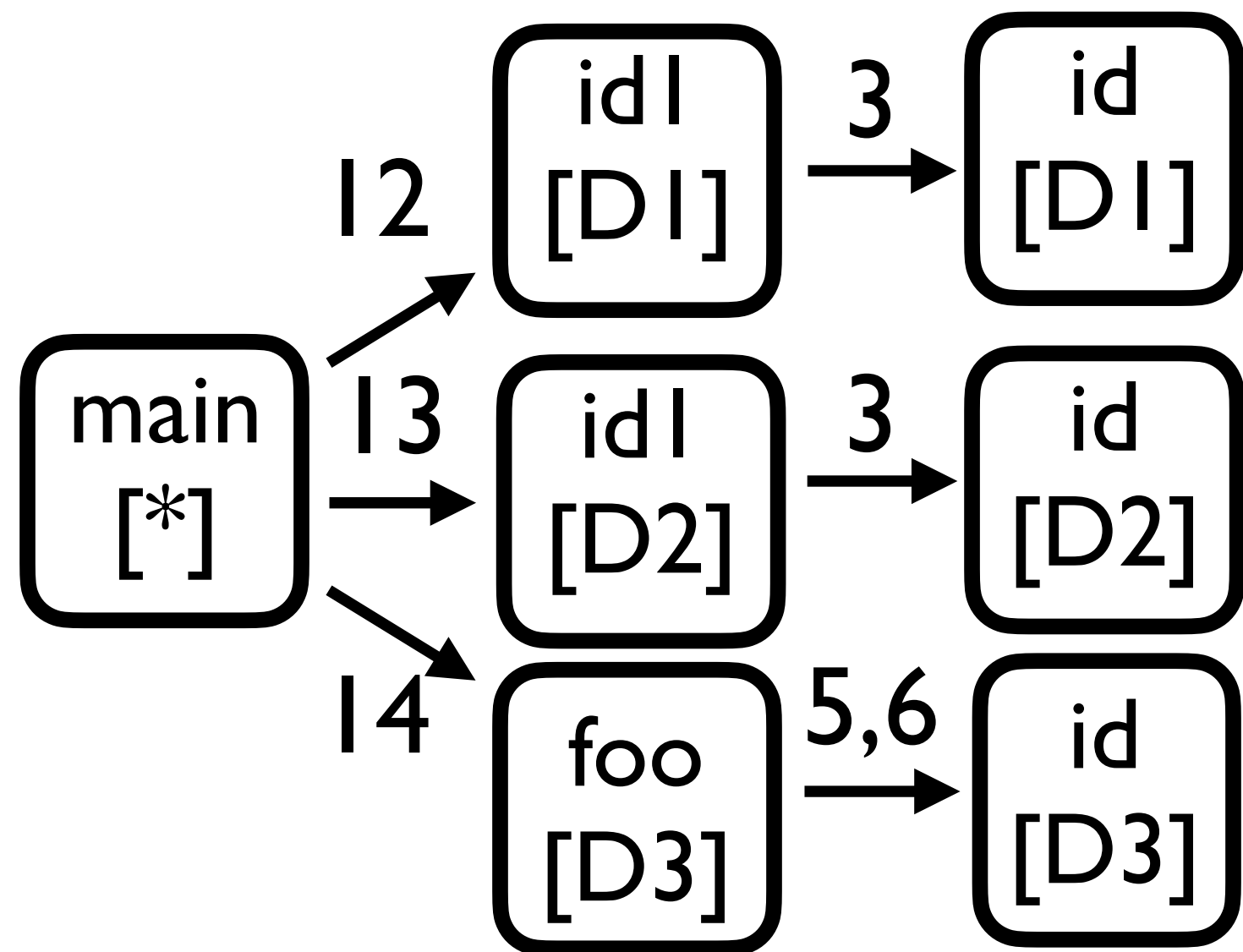
=



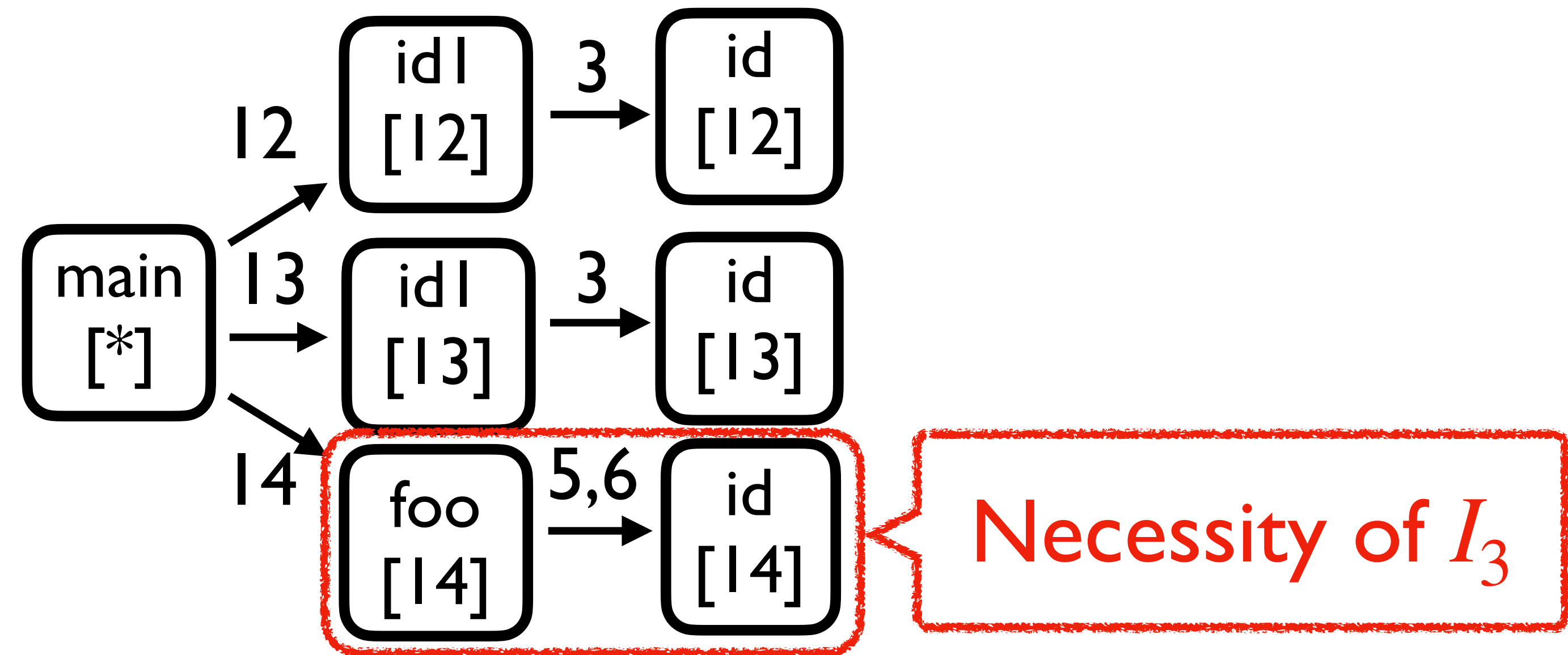
$I_{callH+T'}$  ( $T' = \{3,5,6\}$ )

# Intuition Behind Simulation ( $I_1 \cup I_2$ )

- Suppose given call-graph is produced from  $I_{callH+T}$  and infer what  $T$  is



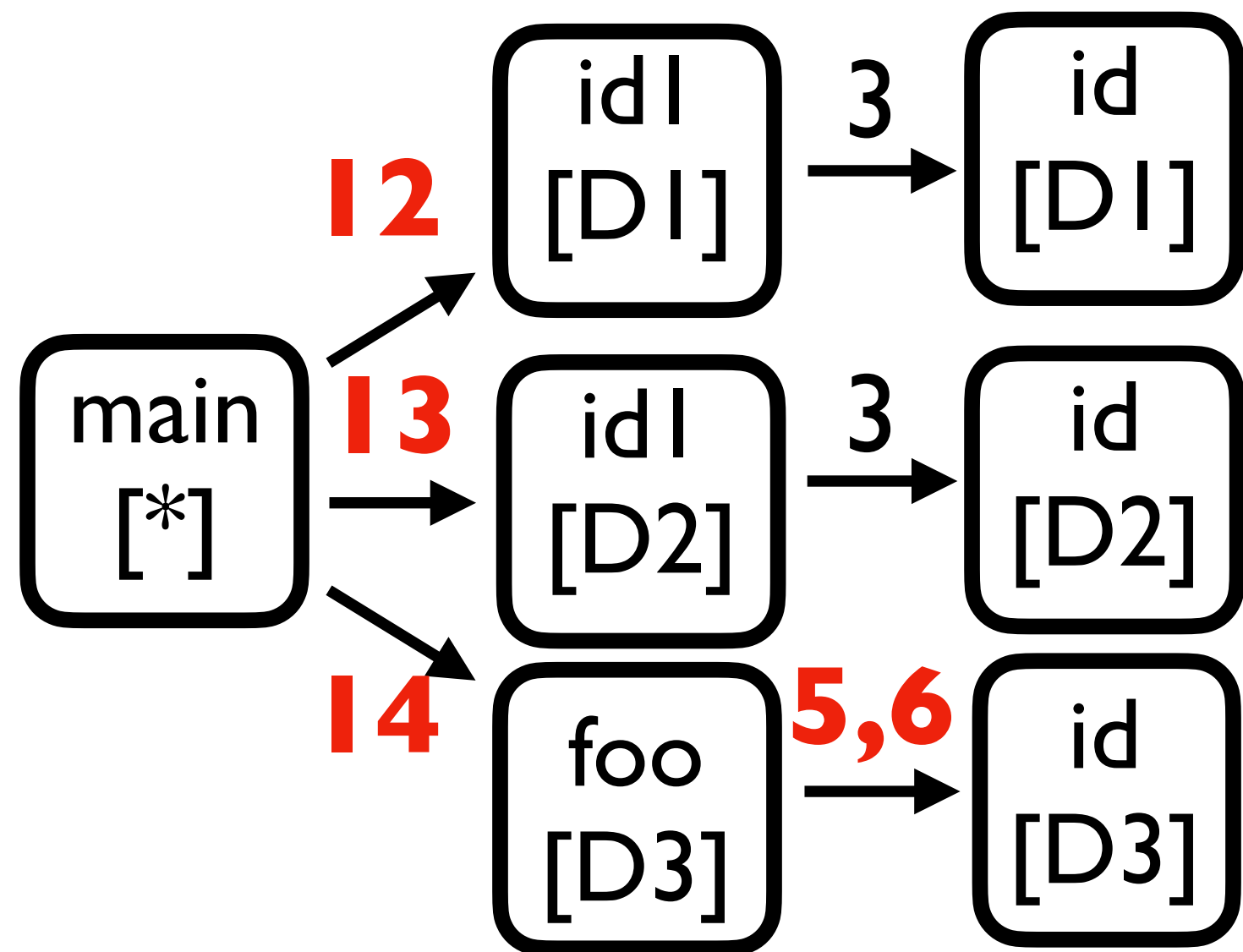
$I_{objH+T}$  ( $T = \emptyset$ )



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

# 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

$$I_2 = \{3\}$$

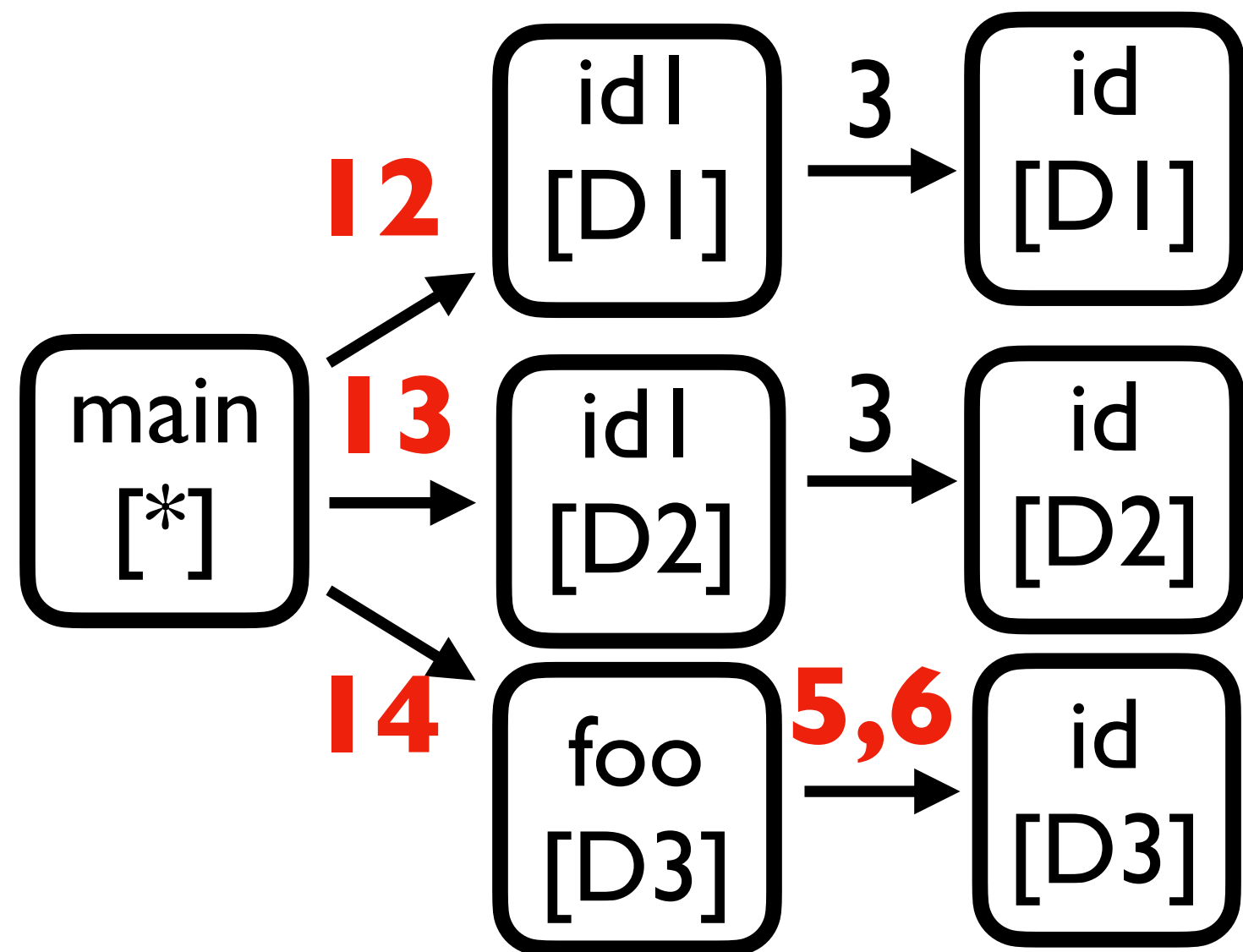
- $I_3$ : given object sensitivity produced only one context

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

$$I_3 = \{5, 6, 12, 13, 14\}$$

# Intuition Behind Simulation

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



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

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

- $I_2$ : different caller ctx imply

$$I_2 = \{3\}$$

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

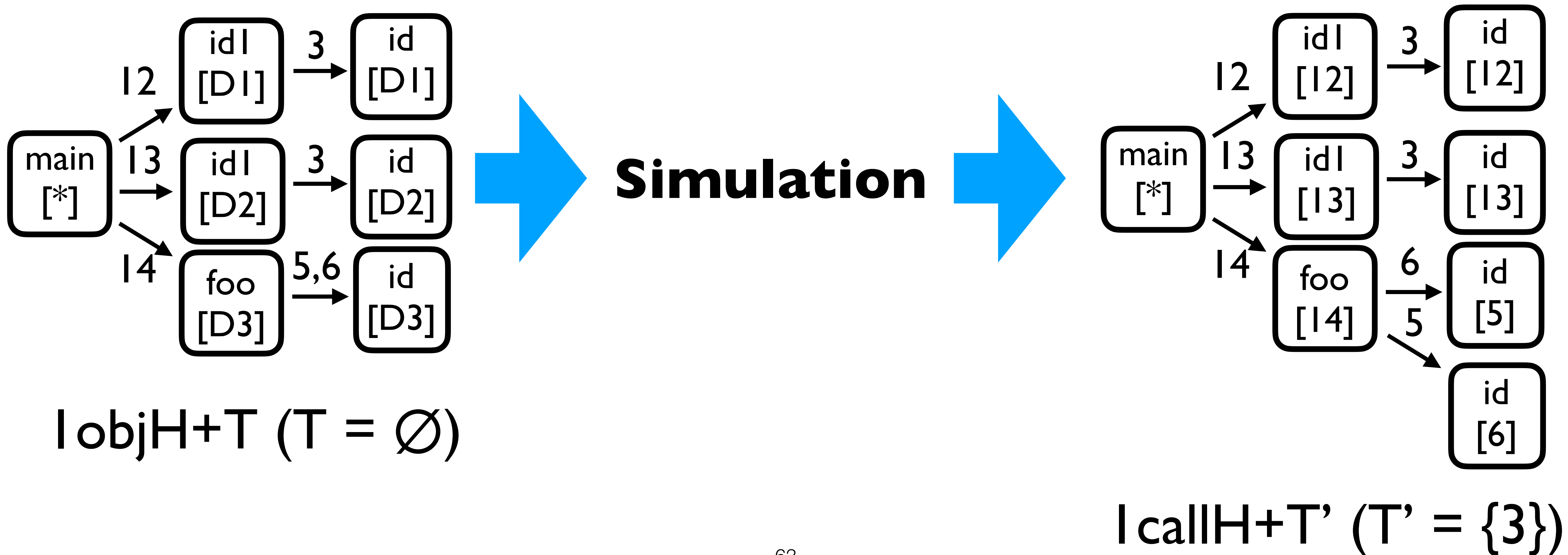
- $I_3$ : given object sensitivity produced only one context

$$I_3 = \{5, 6, 12, 13, 14\}$$

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

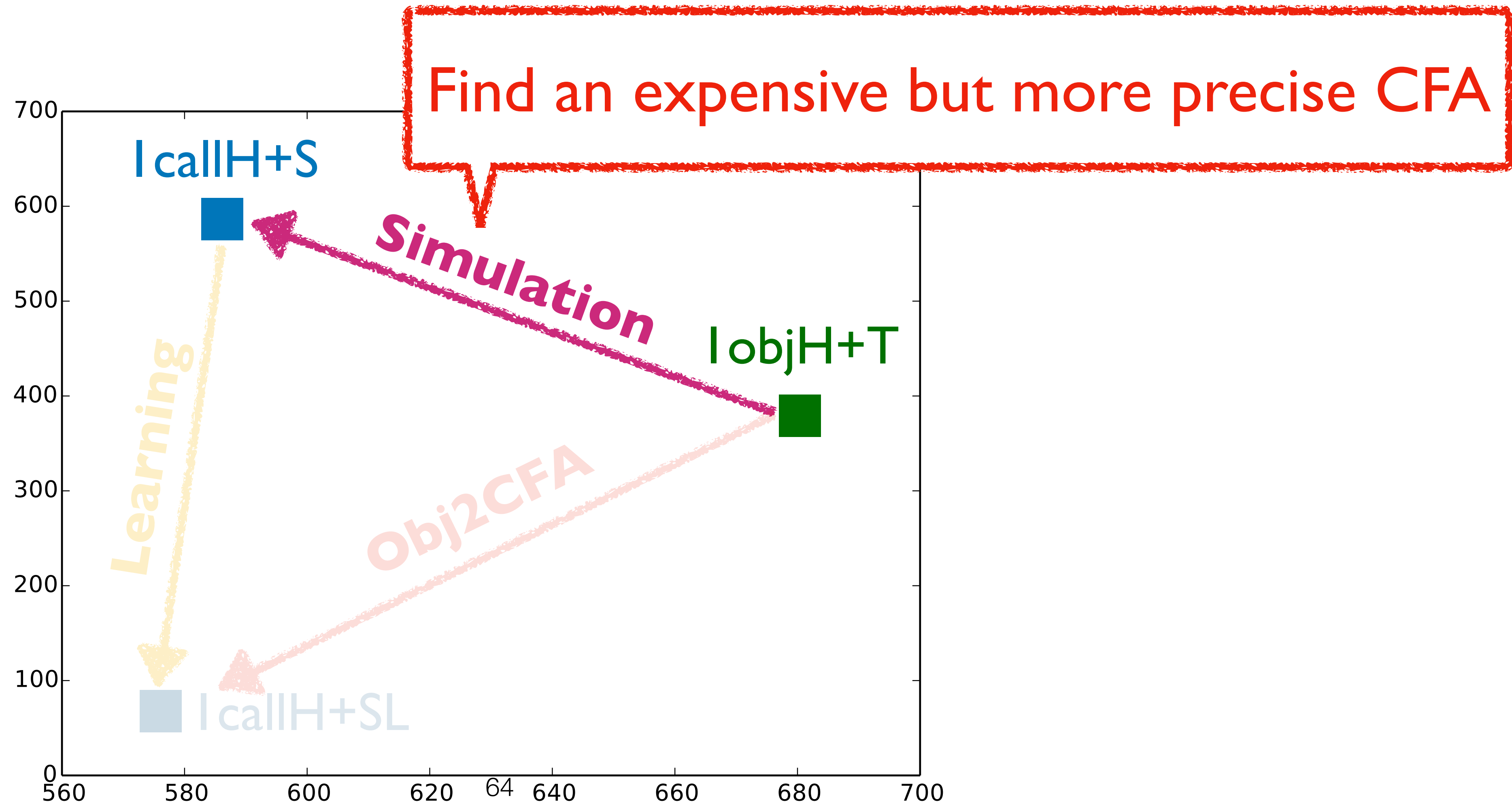
# Technique 1: Simulation

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



# Our Technique : **Obj2CFA**

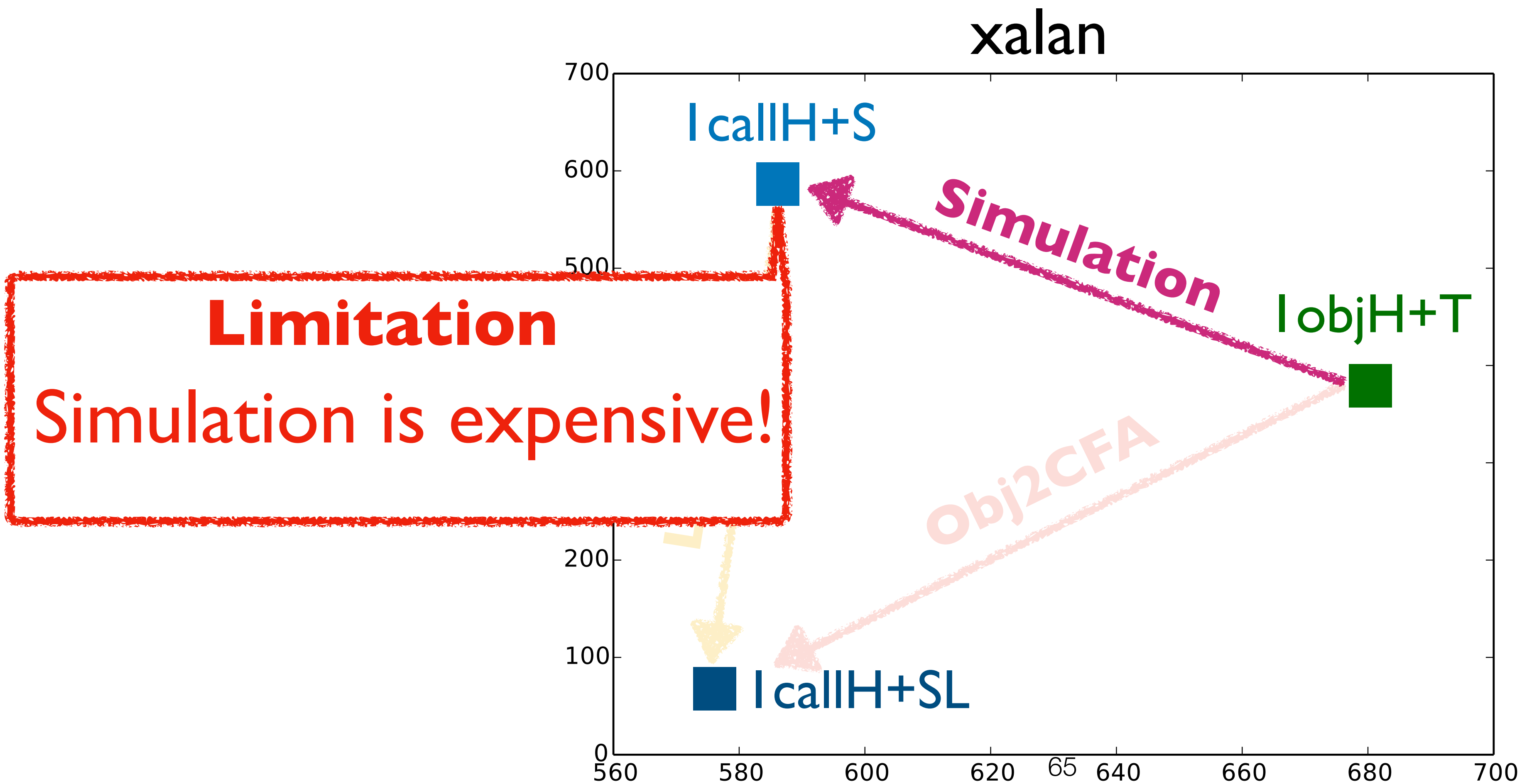
- **Obj2CFA** consists of **simulation** and simulation-guided **learning**





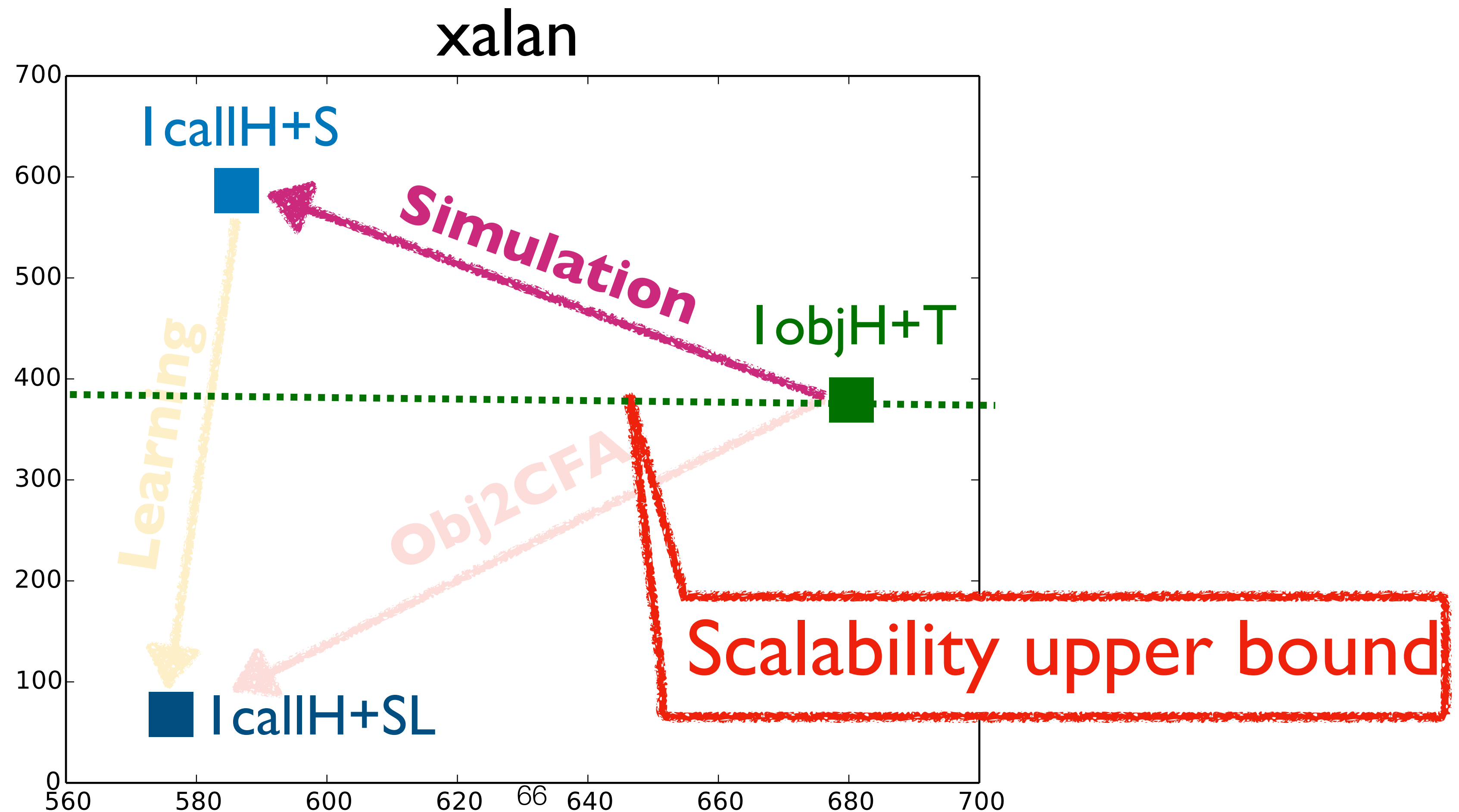
# Our Technique : **Obj2CFA**

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



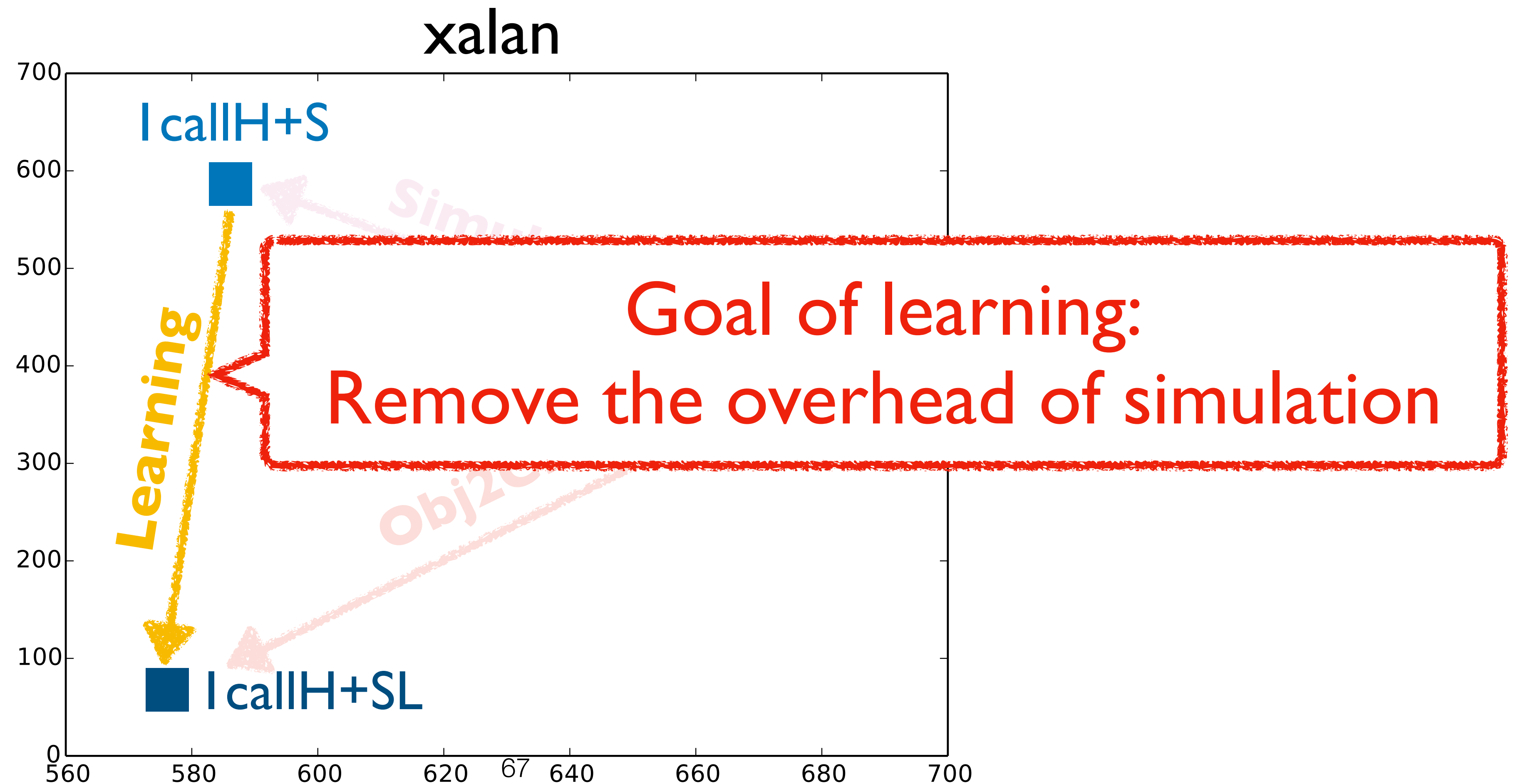
# Our Technique : **Obj2CFA**

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



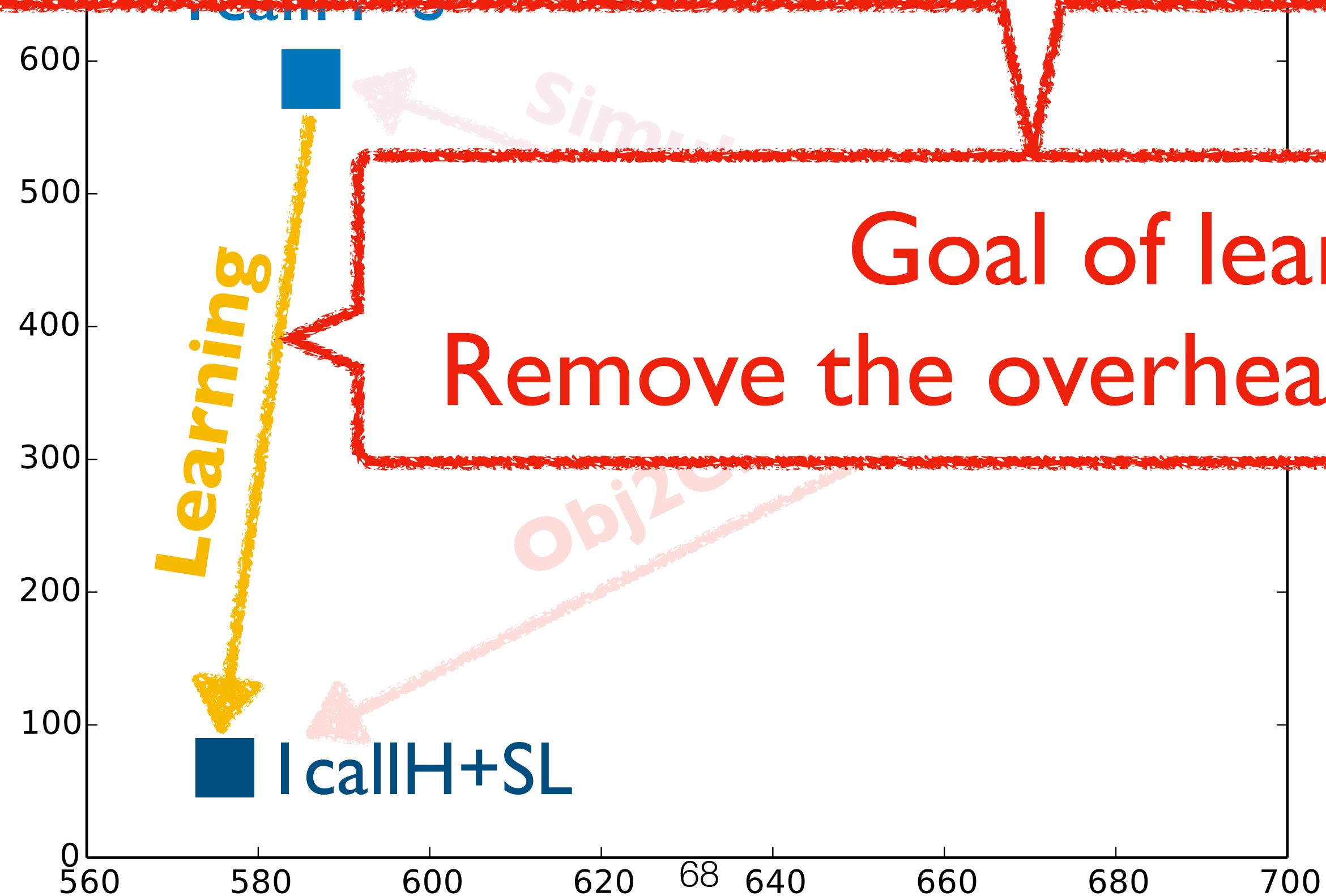
# Our Technique : **Obj2CFA**

- **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

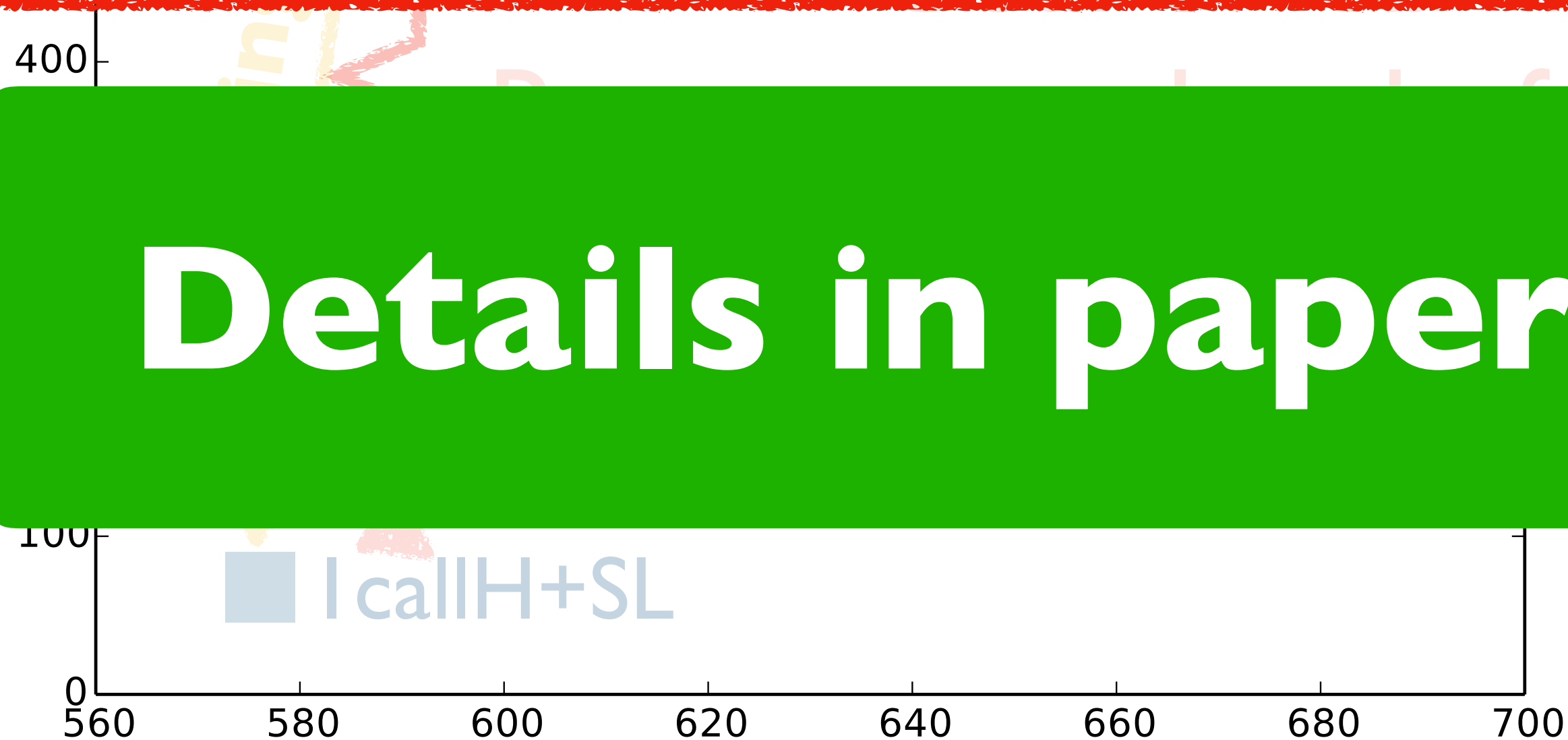


# Our Technique • **Obi2CFA**

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**



# 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

<p><b>Strictly Declarative Specification of Sophisticated Points-to Analysis</b></p> <p>Martin Bravenboer, Yannis Smaragdakis</p> <p>Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA martin.bravenboer@acm.org, yannis@cs.umass.edu</p> <p><b>Abstract</b></p> <p>We present the Doop framework for points-to analysis of Java programs. Doop builds on the idea of specifying pointer analysis algorithms declaratively, using Datalog: a logic-based language for defining (recursive) relations. We carry the declarative approach further than past work by describing the full end-to-end analysis in Datalog and optimizing aggressively using a novel technique specifically targeting highly recursive Datalog programs.</p> <p>As a result, Doop achieves several benefits, including full order-of-magnitude improvements in runtime. We compare Doop with Lhotak and Hendren's Patox, which defines the state of the art for context-sensitive analyses. For the exact same logical points-to definitions (and, consequently, identical precision) Doop is more than 15x faster than Patox for a call-site sensitive analysis of the DaCapo benchmarks, with lower but still substantial speedups for other important analyses. Additionally, Doop scales to very precise analyses that are impossible with Patox and Whaley et al.'s bdbddbb, 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 due to its declarative nature.</p> <p>The main elements of our approach are: a language for specifying the program aggressive optimization of the Datalog program analysis (both low-level [6,9]) is far from new. Our novel approach, however, accounts for several order-of-magnitude improvements: unoptimized run over 1000 times more slowly; Generators fit well the approach of handling the database, by specifically targeting the incremental evaluation of Datalog implementations, our approach is entirely Datalog-declarative; the logic required both for generation as well as for handling the full set of the Java language (e.g., static initialization, reference objects, threads, exceptions, etc.) makes our pointer analysis specifications both also efficient and easy to tune. Generators data point in support of declarative logic that prohibitively much human effort planning and optimizing complex mutations at an operational level of abstract</p>	<p><b>Pick Your Contexts Well: Understanding the Making of a Precise and Scalable Pointer Analysis</b></p> <p>Yannis Smaragdakis, Martin Bravenboer</p> <p>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</p> <p><b>Abstract</b></p> <p>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).</p> <p><b>Categories and Subject Descriptors</b> F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis; D.3.4 [Programming Languages]: Processors—Compilers</p> <p><b>General Terms</b> Algorithms, Languages, Performance</p> <p><b>Keywords</b> points-to analysis; context-sensitivity; object-sensitivity; type-sensitivity</p> <p><b>1. Introduction</b></p> <p>Points-to analysis is a static program analysis that consists of computing all objects (typically identified by allocation sites) 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.</p>	<p><b>Hybrid Context-Sensitivity for Points-To Analysis</b></p> <p>George Kastirinis, Yannis Smaragdakis</p> <p>Department of Informatics, University of Athens, {kastirinis,smaragd}@di.uoa.gr</p> <p><b>Abstract</b></p> <p>Context-sensitive points-to analysis is valuable for achieving high precision with good performance. The standard flavors of context-sensitivity are call-site-sensitivity (KCF) 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 more 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-sensitivity) establishing a new set of performance/precision sweet spots.</p> <p><b>Categories and Subject Descriptors</b> F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis; D.3.4 [Programming Languages]: Processors—Compilers</p> <p><b>General Terms</b> Algorithms, Languages, Performance</p> <p><b>Keywords</b> points-to analysis; context-sensitivity; object-sensitivity; type-sensitivity</p> <p><b>1. Introduction</b></p> <p>Points-to analysis is a static program analysis that consists of computing all objects (typically identified by allocation sites) 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.</p>	<p><b>Introspective Analysis: Context-Sensitivity, Across the Board</b></p> <p>Yannis Smaragdakis, George Kastirinis, George Balatsouras</p> <p>Department of Informatics, University of Athens, {smaragd, kastirinis, gbalats}@di.uoa.gr</p> <p><b>Abstract</b></p> <p>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-detrimental 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 or 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.</p> <p><b>Categories and Subject Descriptors</b> F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program Analysis; D.3.4 [Programming Languages]: Processors—Compilers</p> <p><b>General Terms</b> Algorithms, Languages, Performance</p> <p><b>Keywords</b> points-to analysis; context-sensitivity; object-sensitivity; type-sensitivity</p> <pre>class C {   void foo(Object o) { ... } } class Client {   void bar(C c1, C c2) { ...     ...     c1.foo(obj1);     c2.foo(obj2);   } }</pre> <p>In contrast, object-sensitivity uses ob of instructions containing a new 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 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.</p>	<p><b>Making <math>k</math>-Object-Sensitive Pointer Analysis More Precise with Still <math>k</math>-Limiting</b></p> <p>Tian Tan<sup>1</sup>, Yue Li<sup>1</sup>, and Jingling Xue<sup>1,2</sup></p> <p><sup>1</sup> School of Computer Science and Engineering, UNSW Australia <sup>2</sup> Advanced Innovation Center for Imaging Technology, CNU, China</p> <p><b>Abstract</b></p> <p>Object-sensitivity is regarded as arguably the best context abstraction for pointer analysis in object-oriented languages. However, a <math>k</math>-object-sensitive pointer analysis, which uses a sequence of <math>k</math> allocation sites (as <math>k</math> 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 <math>k</math>-object-sensitive analysis, denoted <math>k</math>-obj, by still using a <math>k</math>-limiting context abstraction. The novelty is to identify allocation sites that are redundant context elements in <math>k</math>-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 <math>k</math>-object-sensitive analysis for the program. BEAN is generally more precise than <math>k</math>-obj, with a precision that is guaranteed to be as good as <math>k</math>-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 (<i>may-alias</i> and <i>may-fail-cast</i>) 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.</p>	<p><b>Data-Driven Context-Sensitivity for Points-to Analysis</b></p> <p>SEHUN JEONG, Korea University, Republic of Korea MINSEOK JEON, Korea University, Republic of Korea SUNGDEOK CHA, Korea University, Republic of Korea HAKJOO OH<sup>1</sup>, Korea University, Republic of Korea</p> <p>We present a new data-driven approach to achieve highly cost-effective context-sensitive points-to analysis for Java. While context-sensitivity has greater impact on the analysis precision and performance than any other 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 designing 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 and 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, experimental results show that our approach significantly outperforms existing techniques.</p> <p>CCS Concepts: • Theory of computation → Program analysis; • Computing methodologies → Machine learning approaches;</p> <p>Additional Key Words and Phrases: Data-driven program analysis, Points-to analysis, Context-sensitivity</p> <p>ACM Reference Format: Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-Driven Context-Sensitivity for Points-to Analysis. <i>Proc. ACM Program. Lang.</i> 1, OOPSLA, Article 100 (October 2017), 27 pages. <a href="https://doi.org/10.1145/3133924">https://doi.org/10.1145/3133924</a></p> <p><b>1 INTRODUCTION</b></p> <p>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 of subsequent higher-level program analyses such as static bug-finders, security auditing tools, and program understanding tools.</p> <p>For object-oriented languages, context-sensitive points-to analysis is important as it must distinguish method's local variables and objects in different calling-contexts. For languages like Java, the first and second authors contributed equally to this work <sup>1</sup>Corresponding author</p> <p>Authors' email addresses: S. Jeong, gifrang@korea.ac.kr; M. Jeon, minseok_jeon@korea.ac.kr; S. Cha, schai@korea.ac.kr; H. Oh, hakjoo_oh@korea.ac.kr</p> <p>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. Request permissions from permissions@acm.org.</p> <p>© 2017 Association for Computing Machinery. 2475-1421/2017/10-ART100 <a href="https://doi.org/10.1145/3133924">https://doi.org/10.1145/3133924</a></p>
--	---	---	---	---	--

2009 (OOPSLA)

2011 (POPL)

2013 (PLDI)

2014 (PLDI)

2016 (SAS)

2017 (OOPSLA)



# Setting

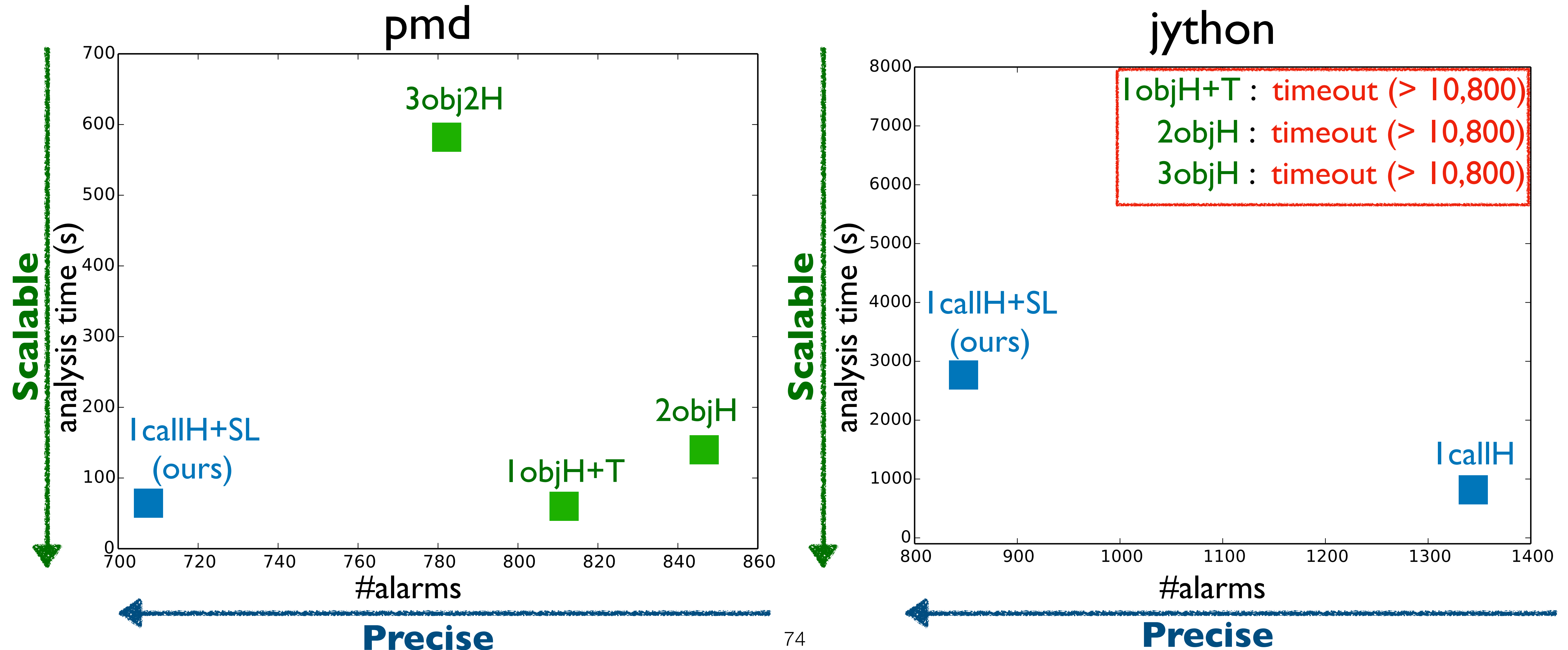
- Doop
- Pointer analysis framework for Java
- Research Question: which one is better?

Call-site sensitivity vs Object sensitivity

Context tunneling is included

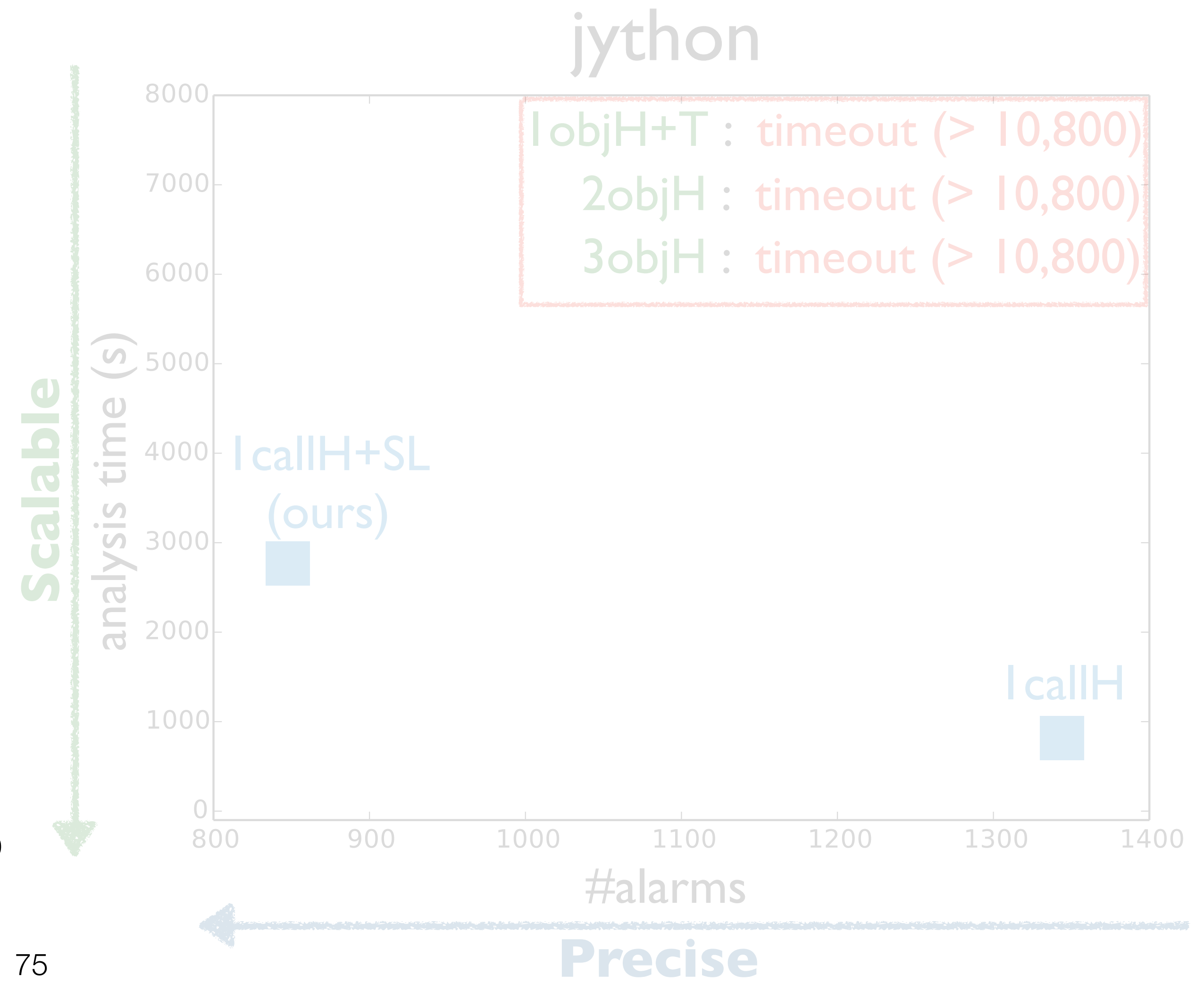
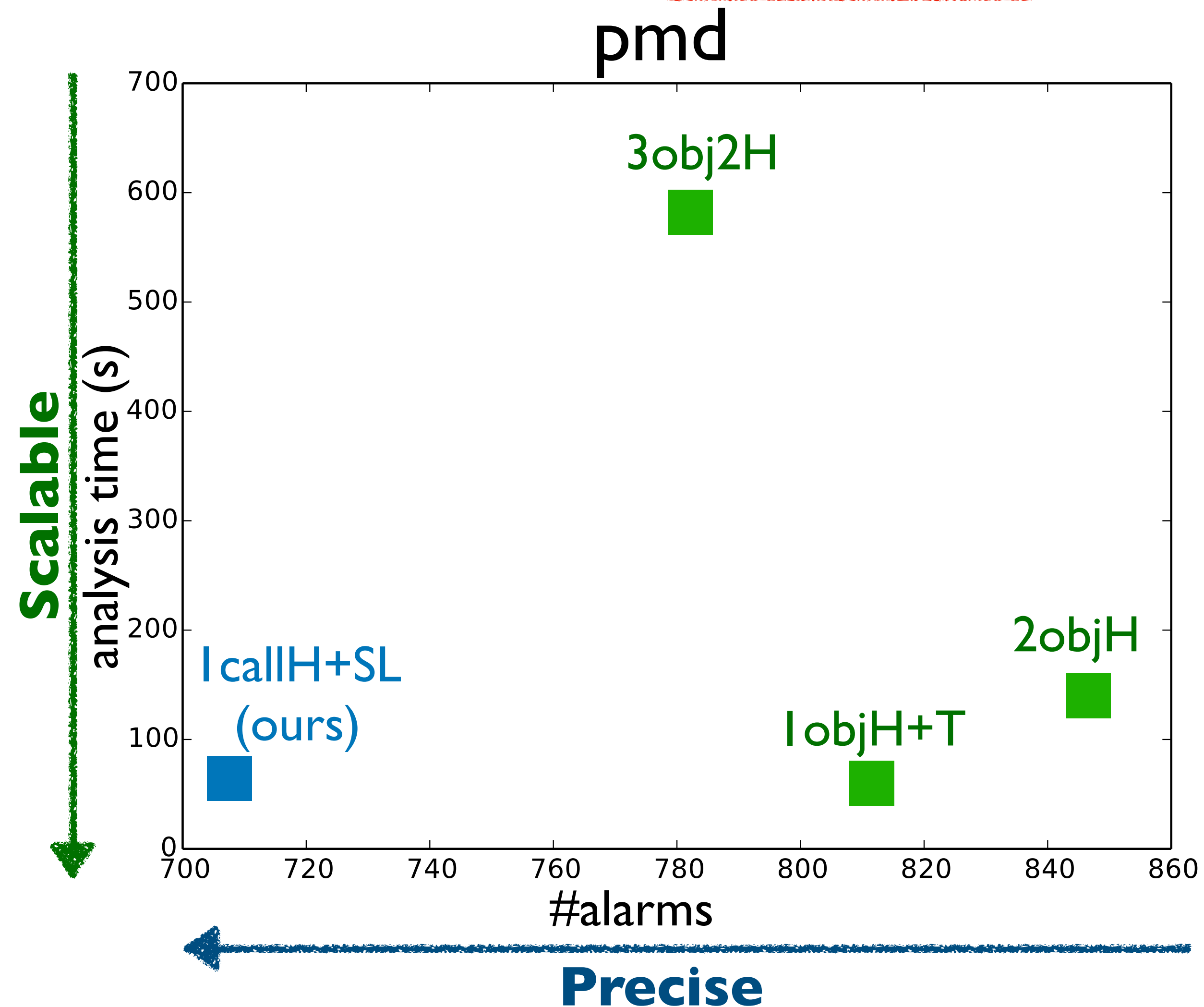
# Call-site Sensitivity vs Object Sensitivity

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



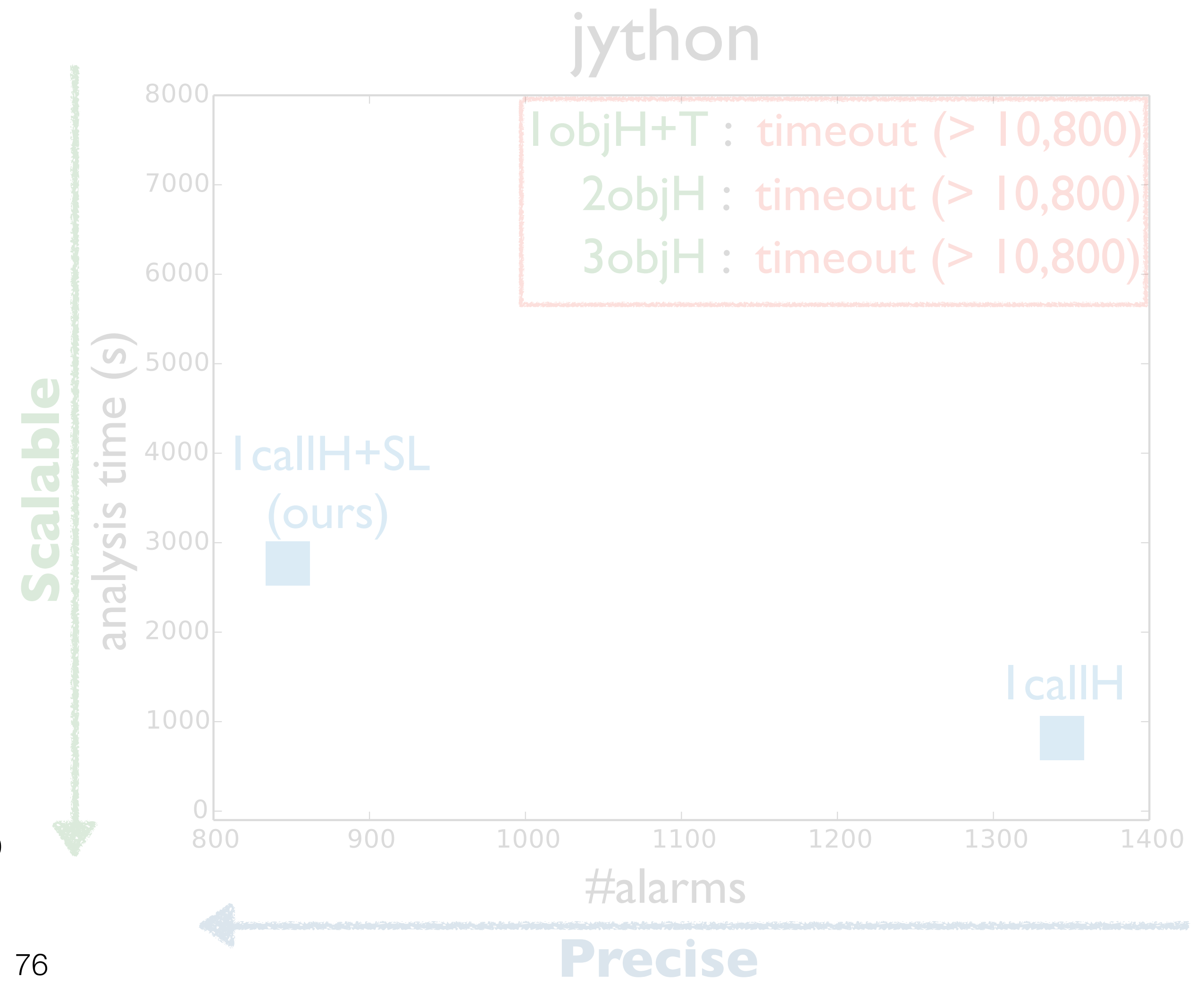
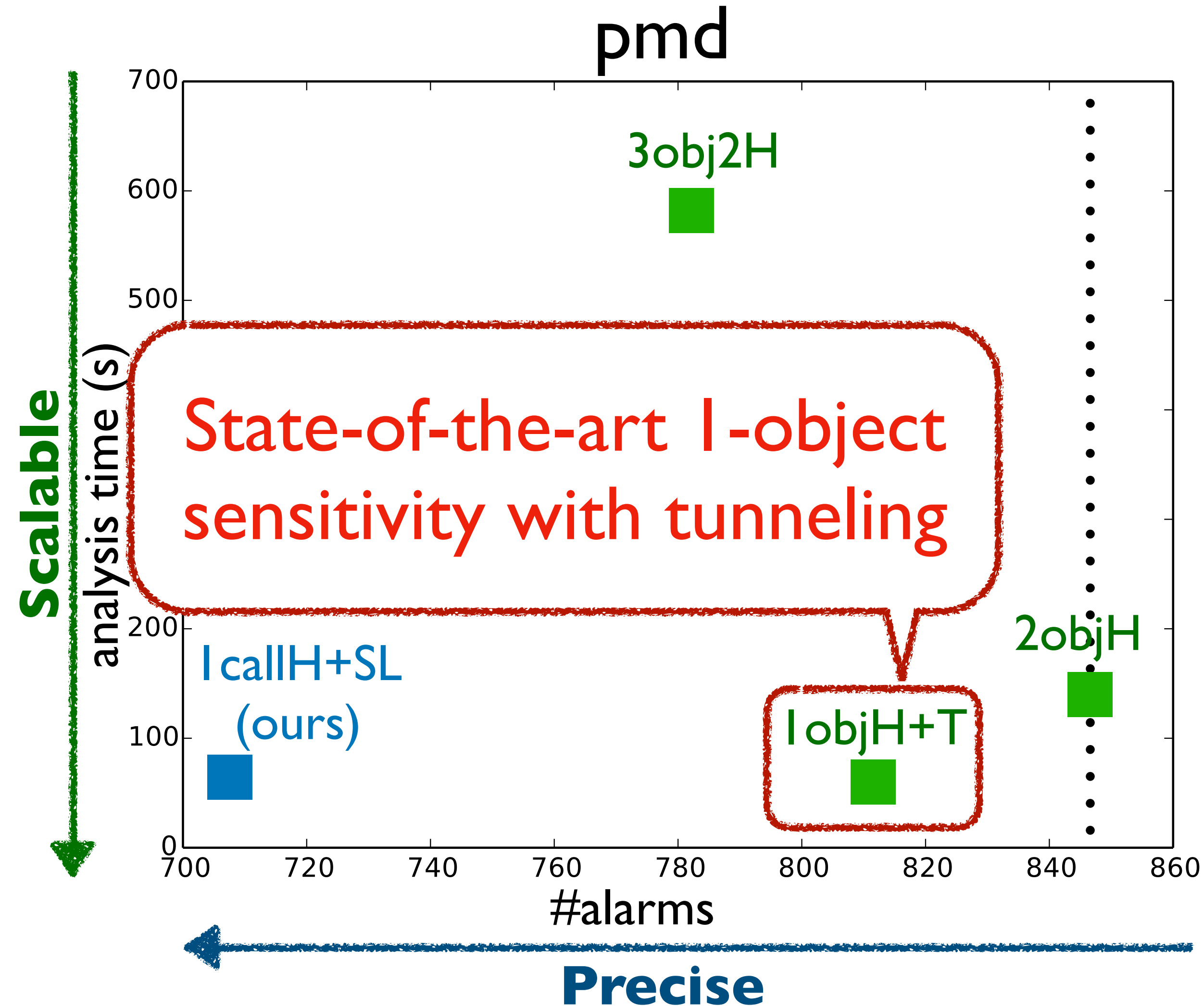
# Call-site Sensitivity vs Object Sensitivity

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



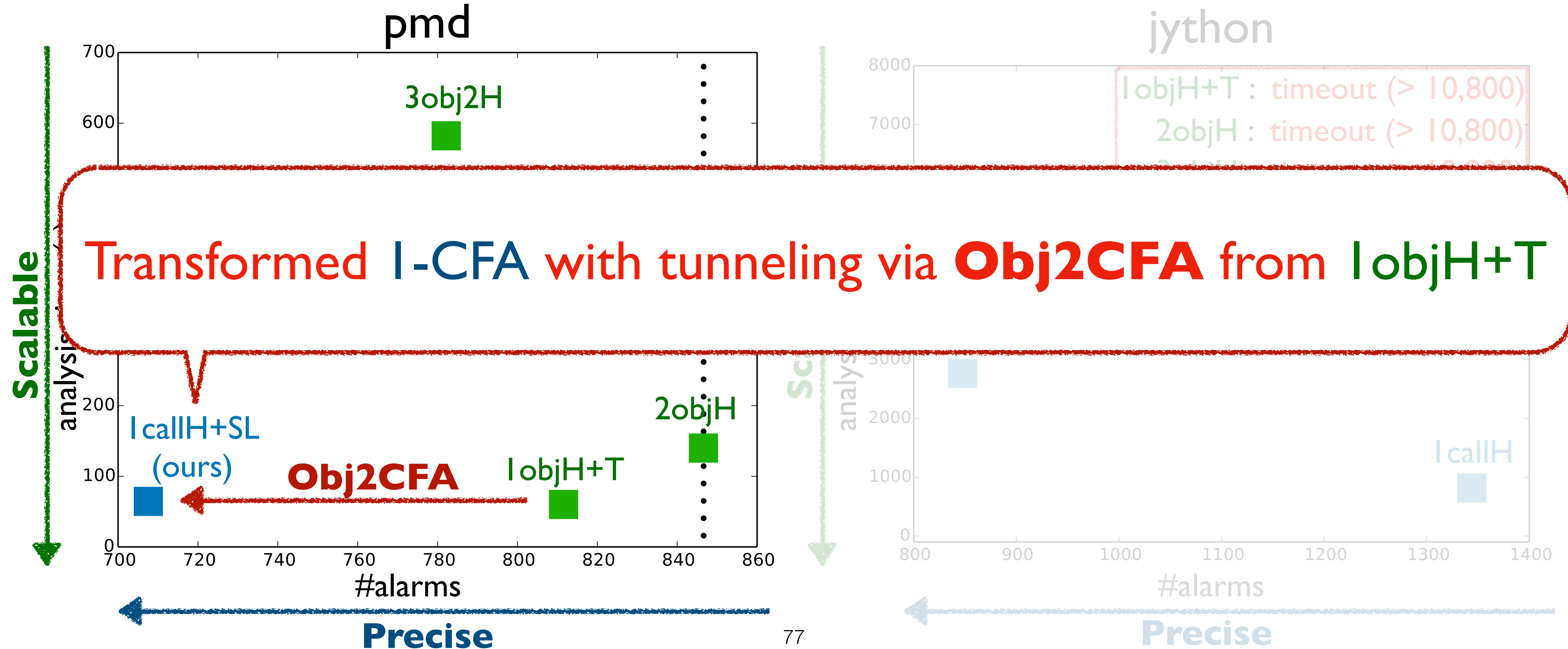
# Call-site Sensitivity vs Object Sensitivity

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



# Call-site Sensitivity vs Object Sensitivity

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

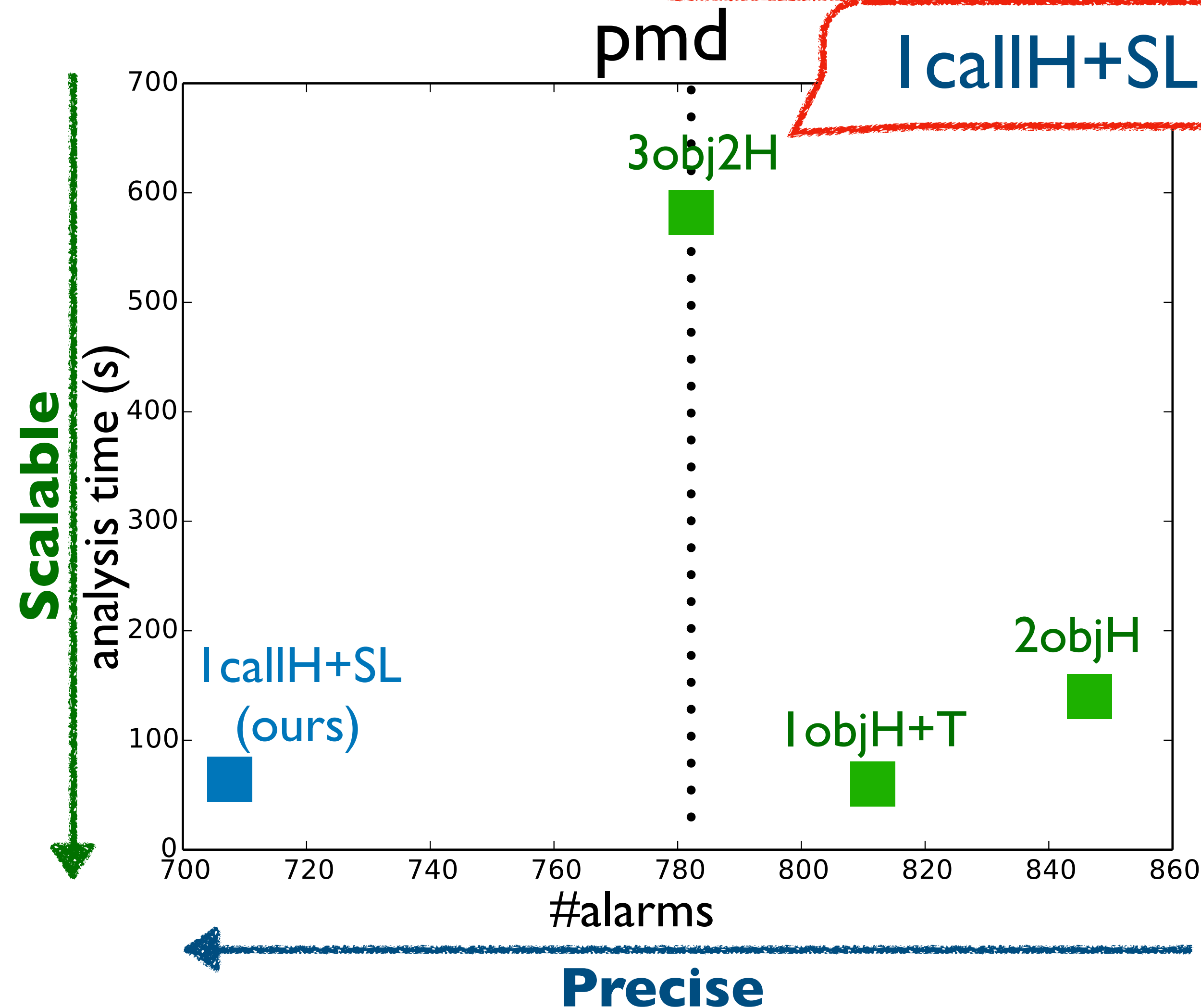


# Call-site Sensitivity vs Object Sensitivity

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

IcallH+SL is **even more precise** than 3obj2H

Precision upper bound of recent researches on **object sensitivity**



**Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity**

TIAN TAN, Nanjing University, China  
YUE LI\*, Nanjing University, China  
XIAOXING MA, Nanjing University, China  
CHANG XU, Nanjing University, China  
YANNIS SMARAGDAKIS, University of Athens, Greece

Traditional context-sensitive pointer analysis is hard to scale for large and complex Java programs. To address this issue, a series of selective context-sensitivity approaches have been proposed and exhibit promising results. In this work, we move one step further towards producing highly-precise pointer analyses for hard-to-analyze Java programs by presenting the Unity-Relay framework, which takes selective context sensitivity to the next level. Briefly, Unity-Relay is a one-two punch: given a set of different selective context-sensitivity approaches, say  $S = S_1, \dots, S_n$ , Unity-Relay first provides a mechanism (called Unity) to combine and maximize the precision of all components of  $S$ . When Unity fails to scale, Unity-Relay offers a scheme (called Relay) to pass and accumulate the precision from one approach  $S_i$  in  $S$  to the next,  $S_{i+1}$ , leading to an analysis that is more precise than all approaches in  $S$ .

As a proof-of-concept, we instantiate Unity-Relay into a tool called Barton and extensively evaluate it on a set of hard-to-analyze Java programs, using general precision metrics and popular clients. Compared with the state of the art, Barton achieves the best precision for all metrics and clients for all evaluated programs. The difference in precision is often dramatic—up to 71% of alias pairs reported by previously-best algorithms are found to be spurious and eliminated.

CCS Concepts • Theory of computation → Program analysis.

Additional Key Words and Phrases: Pointer Analysis, Alias Analysis, Context Sensitivity, Java

ACM Reference Format:  
Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 147 (October 2021), 27 pages. <https://doi.org/10.1145/3485524>

1 INTRODUCTION  
Pointer analysis is important for an array of real-world applications such as bug detection [Chandra et al. 2009; Naik et al. 2006], security analysis [Arzt et al. 2014; Livshits and Lam 2003], program verification [Fink et al. 2008; Pradel et al. 2012] and program understanding [Li et al. 2016; Sridharan et al. 2017].

Authors' addresses: Tian Tan, State Key Laboratory for Novel Software Technology, Nanjing University, China, tiantan@nju.edu.cn; Yue Li, State Key Laboratory for Novel Software Technology, Nanjing University, China, yueli@nju.edu.cn; Xiaoxing Ma, State Key Laboratory for Novel Software Technology, Nanjing University, China, xma@nju.edu.cn; Chang Xu, State Key Laboratory for Novel Software Technology, Nanjing University, China, changxu@nju.edu.cn; Yannis Smaragdakis, Department of Informatics and Telecommunications, University of Athens, Greece, yannis@smaragdakis.org.

\*Corresponding author

OOPSLA 2021

**Precision-Preserving Yet Fast Object-Sensitive Pointer Analysis with Partial Context Sensitivity**

JINGBO LU, UNSW Sydney, Australia  
JINGLING XUE, UNSW Sydney, Australia

Object-sensitivity is widely used as a context abstraction for computing the points-to information context-sensitively for object-oriented languages like Java. Due to the combinatorial explosion of contexts in large programs,  $k$ -object-sensitive pointer analysis (under  $k$ -limiting), denoted  $k$ -obj, is scalable only for small values of  $k$ , where  $k \leq 2$  typically. A few recent solutions attempt to improve its efficiency by instructing  $k$ -obj to analyze only some methods in the program context-sensitively, determined heuristically by a pre-analysis. While already effective, these heuristics-based pre-analyses do not provide precision guarantees, and consequently, are limited in the efficiency gains achieved. We introduce a radically different approach, *Eagle*, that makes  $k$ -obj run significantly faster than the prior art while maintaining its precision. The novelty of *Eagle* is to enable  $k$ -obj to analyze a method with partial context-sensitivity, i.e., context-sensitively for only some of its selected variables/allocation sites. *Eagle* makes these selections during a lightweight pre-analysis by reasoning about context-free language (CFL) reachability at the level of variables/objects in the program, based on a new CFL-reachability formulation of  $k$ -obj. We demonstrate the advances made by *Eagle* by comparing it with the prior art in terms of a set of popular Java benchmarks and applications.

CCS Concepts • Theory of computation → Program analysis.

Additional Key Words and Phrases: Pointer Analysis, Object Sensitivity, CFL Reachability

ACM Reference Format:  
Jingbo Lu and Jingling Xue. 2019. Precision-Preserving Yet Fast Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 148 (October 2019), 29 pages. <https://doi.org/10.1145/3360574>

1 INTRODUCTION  
For object-oriented languages such as Java, context-sensitivity is known to provide highly useful precision for pointer analysis [Jhotik and Henslers 2008; Smaragdakis et al. 2011]. A context-insensitive pointer analysis, such as Andersen's analysis [Andersen 1994], analyzes a method only once, producing one points-to set for every variable and one abstract object for modeling every allocation site in the method. In contrast, its context-sensitive counterpart analyzes a method multiple times under different calling contexts that abstract its different run-time invocations, thereby producing multiple points-to sets for every variable (with one per context) and multiple abstract objects for modeling every allocation site (with one per context) in the method.

To tame the combinatorial explosion of calling contexts, a context is usually represented by a sequence of  $k$  context elements, under  $k$ -limiting. There are two representative abstractions for a method-oriented program: (1)  $k$ -callsite-sensitivity [Shavers 1991], which distinguishes the contexts of a method by its  $k$ -most-recent callsites, and (2)  $k$ -object-sensitivity [Milanova et al. 2005], which

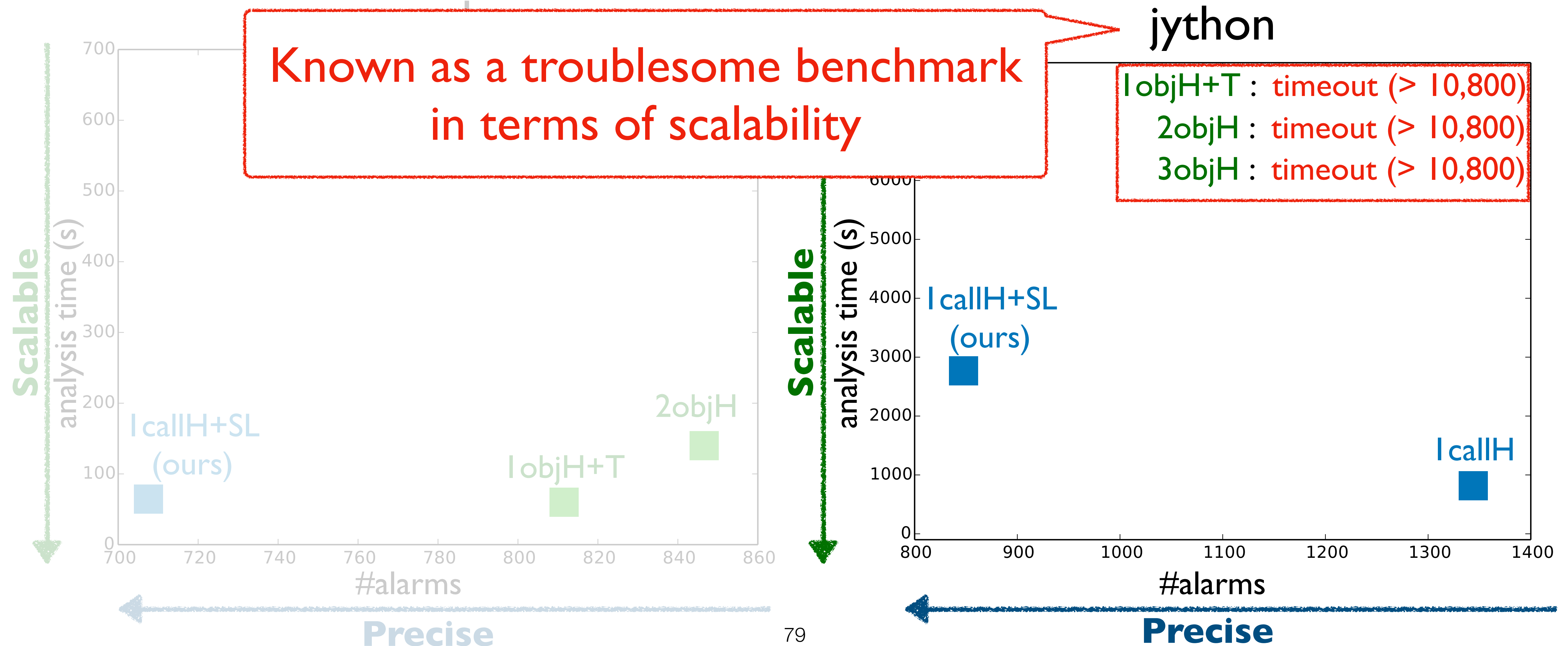
Authors' addresses: Jingbo Lu, UNSW Sydney, Australia, jlu@ese.unsw.edu.au; Jingling Xue, UNSW Sydney, Australia, jingling@ese.unsw.edu.au

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.  
© 2019 Copyright held by the owner/author(s).  
2475-1421/2019/10-ART148  
<https://doi.org/10.1145/3360574>

OOPSLA 2019

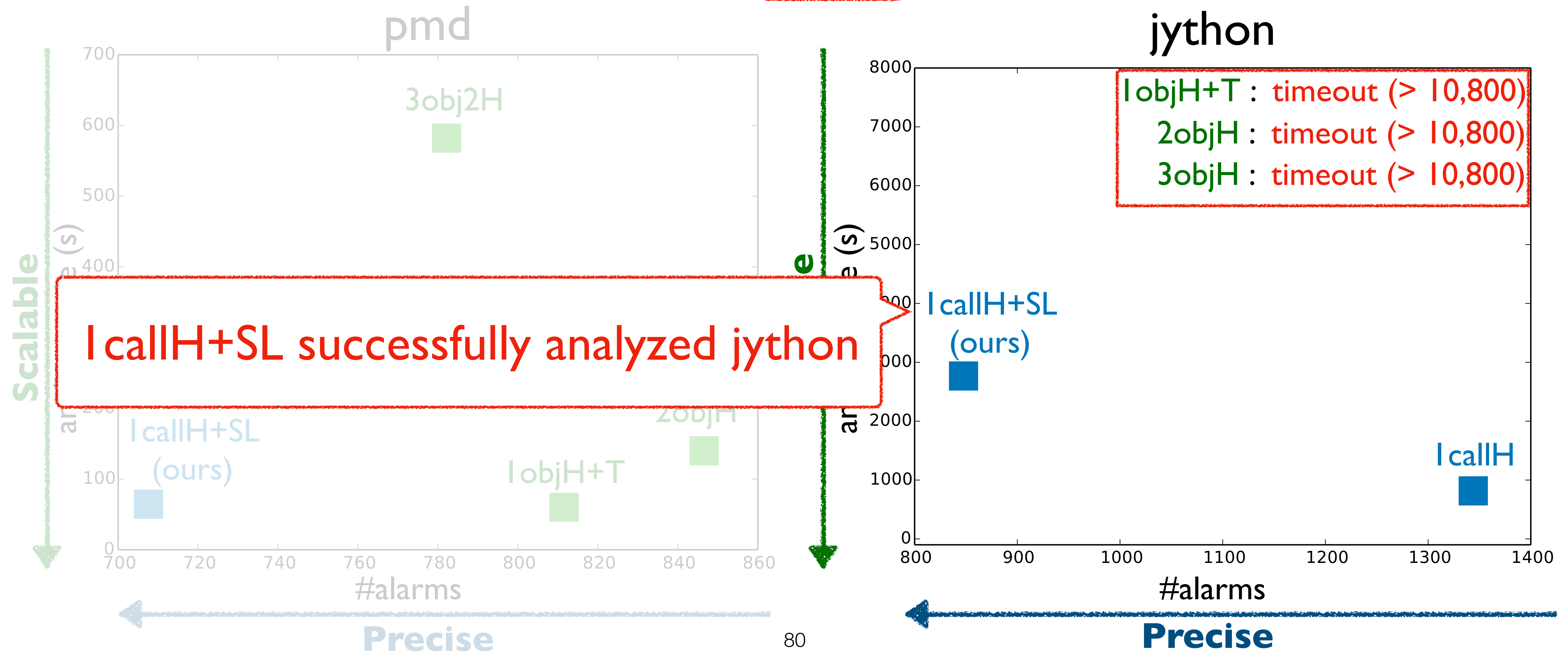
# Call-site Sensitivity vs Object Sensitivity

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



# Call-site Sensitivity vs Object Sensitivity

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

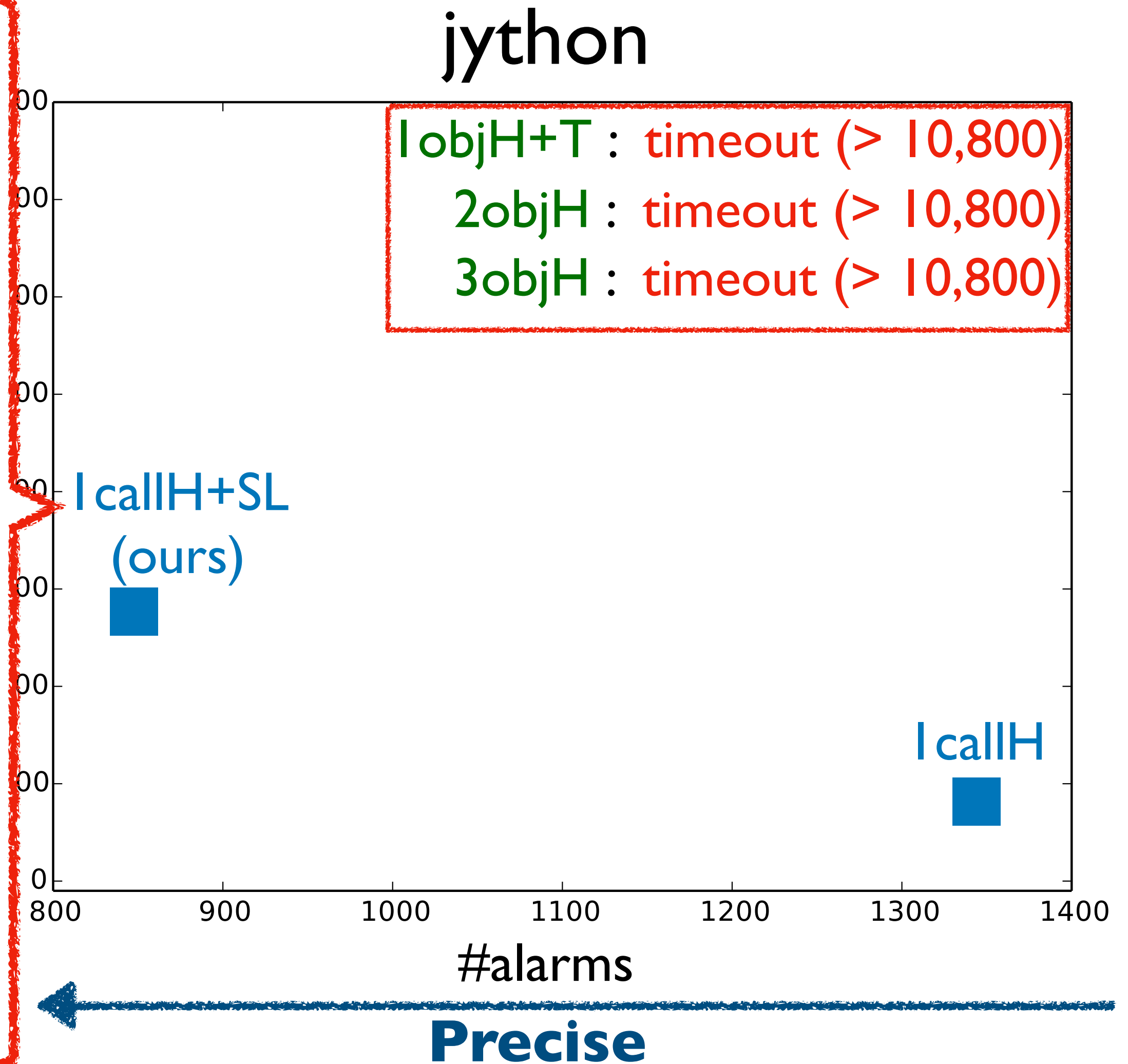
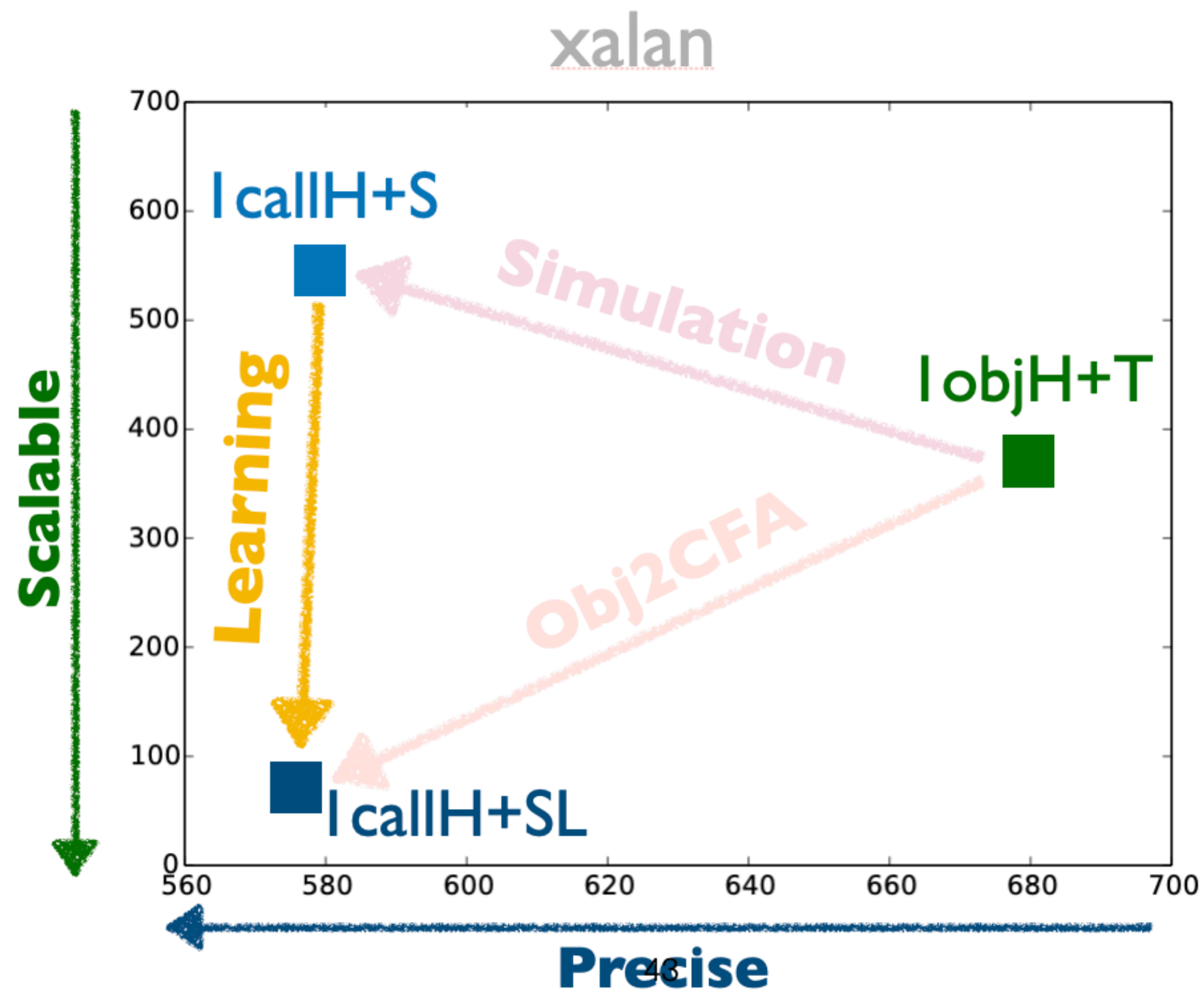




# Call-site Sensitivity vs Object Sensitivity

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

- Necessity of learning
- `IcallH+S` is unable to analyze `xython`



# 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

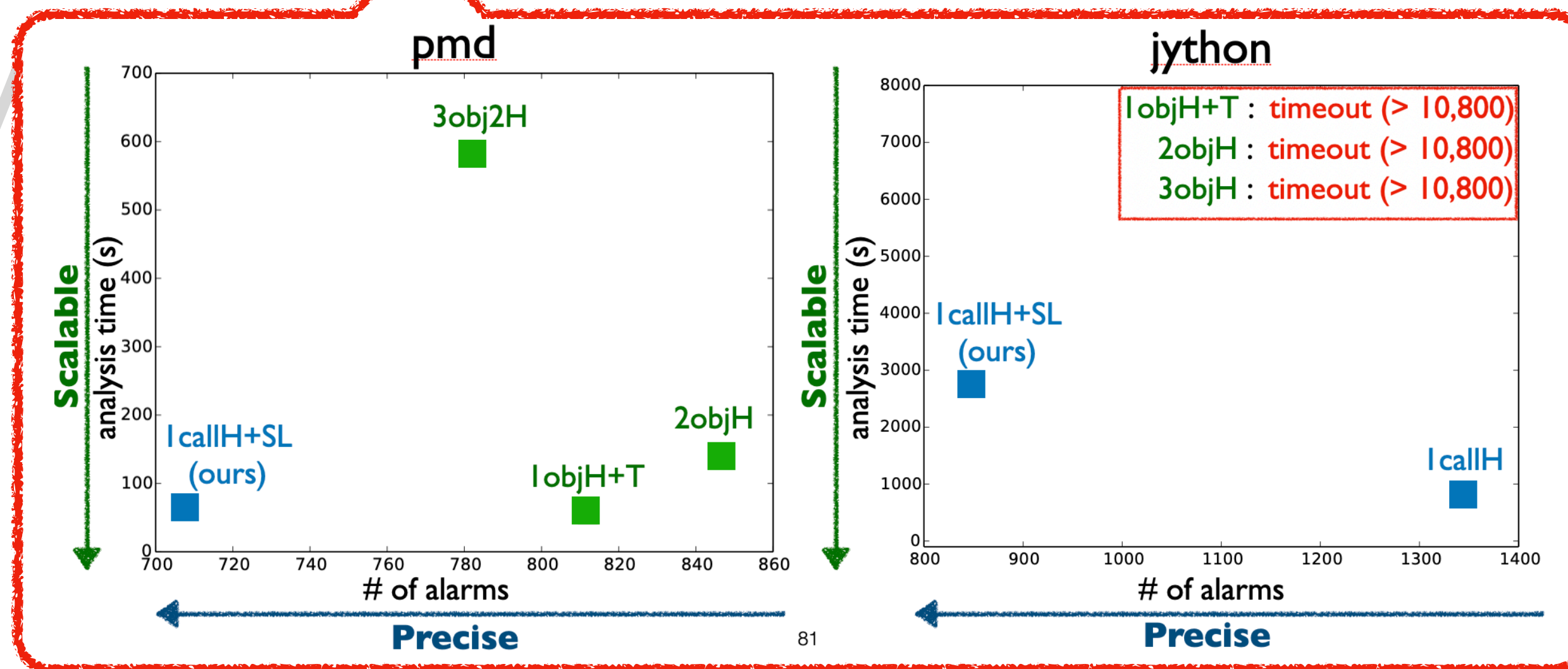
2010

2022

# Summary

- Currently, CFA is known as a bad context

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



- W

om 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  $fob$ . 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