

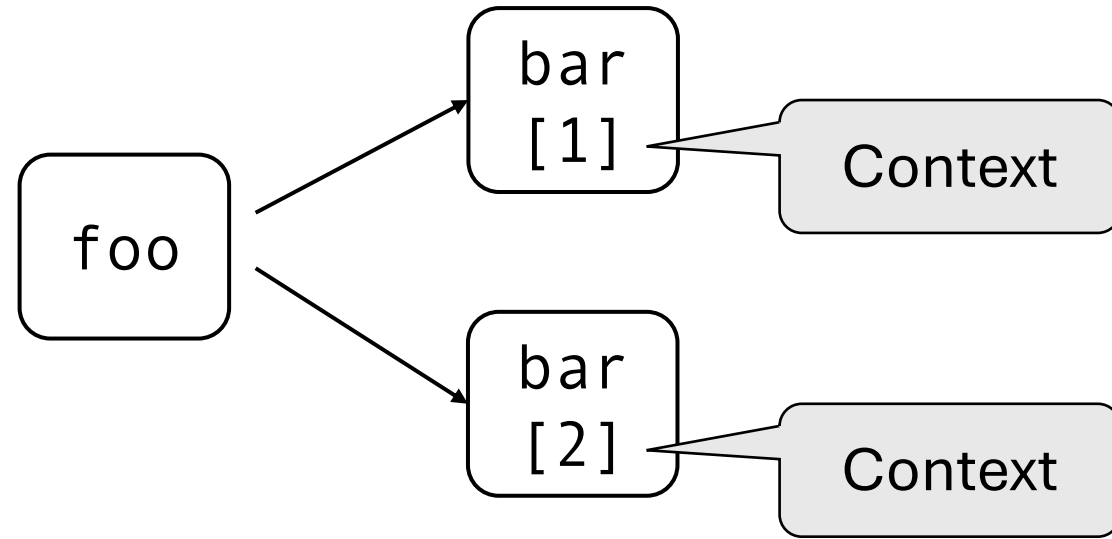
Return of CFA:  
**Call-Site Sensitivity Can Be Superior**  
**to Object Sensitivity**  
Even for Object-Oriented Programs

# Call-Site Sensitivity vs Object Sensitivity

# Call-Site Sensitivity vs Object Sensitivity

Call-Site Sensitivity considers "where"

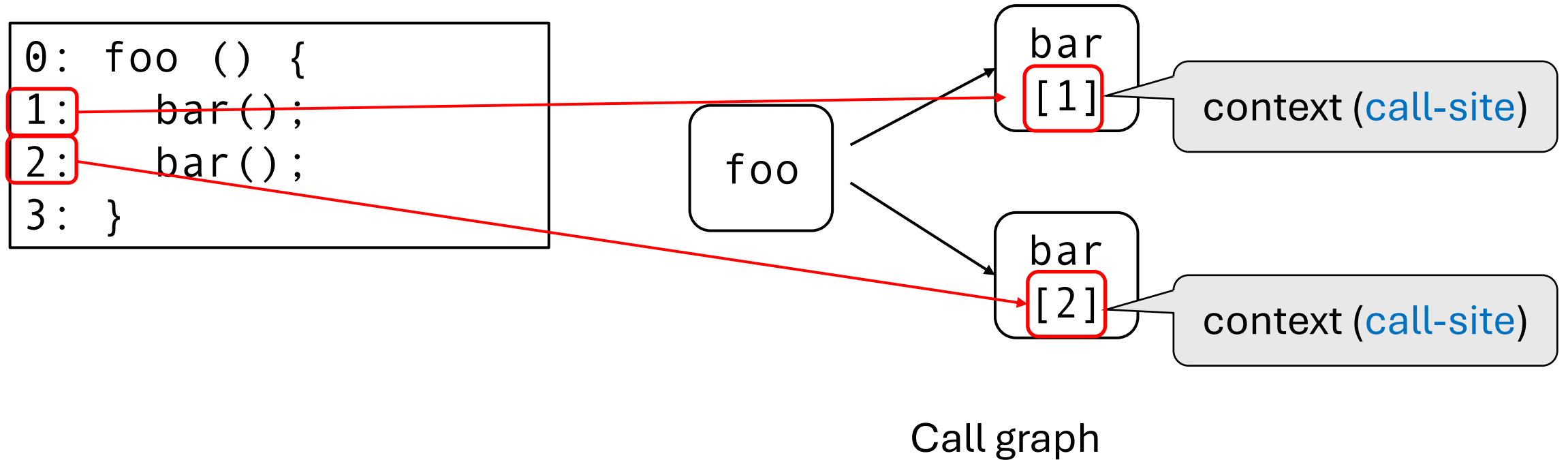
```
0: foo () {  
1:   bar();  
2:   bar();  
3: }
```



Call graph

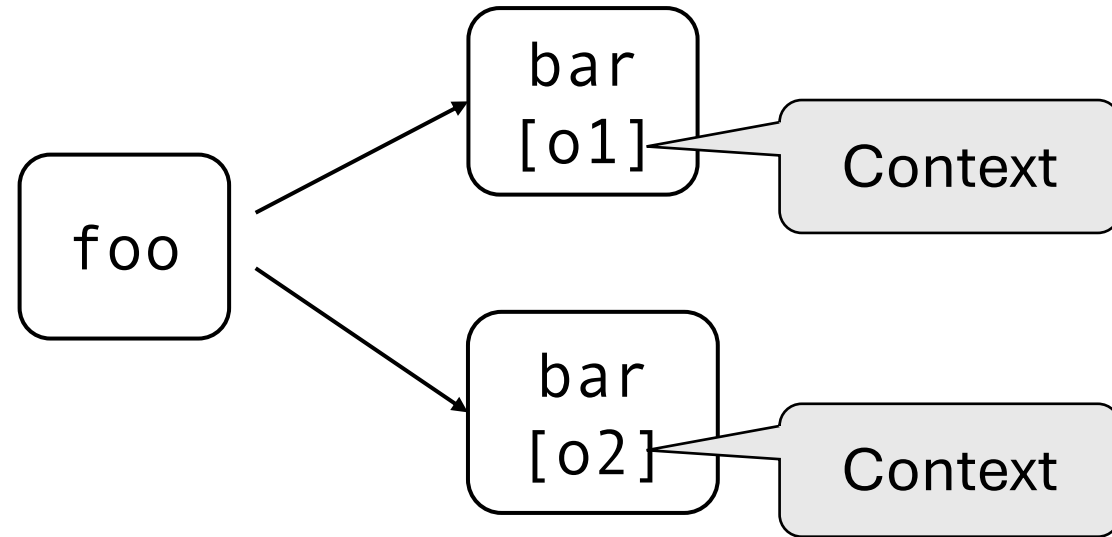
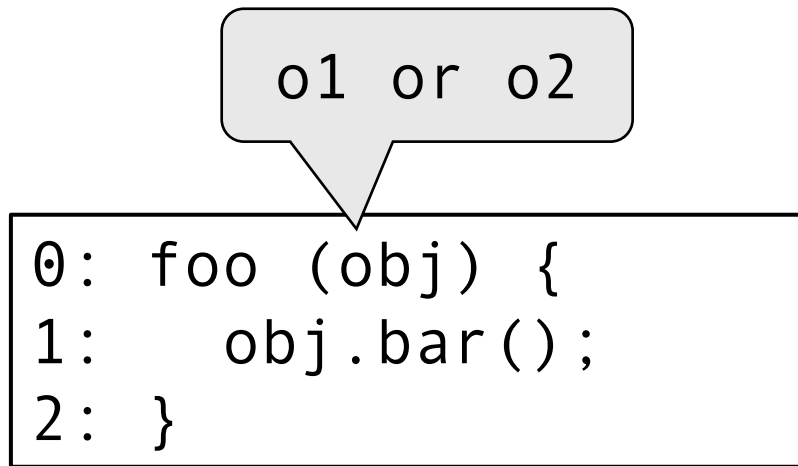
# Call-Site Sensitivity vs Object Sensitivity

Call-Site Sensitivity considers "where"



# Call-Site Sensitivity vs Object Sensitivity

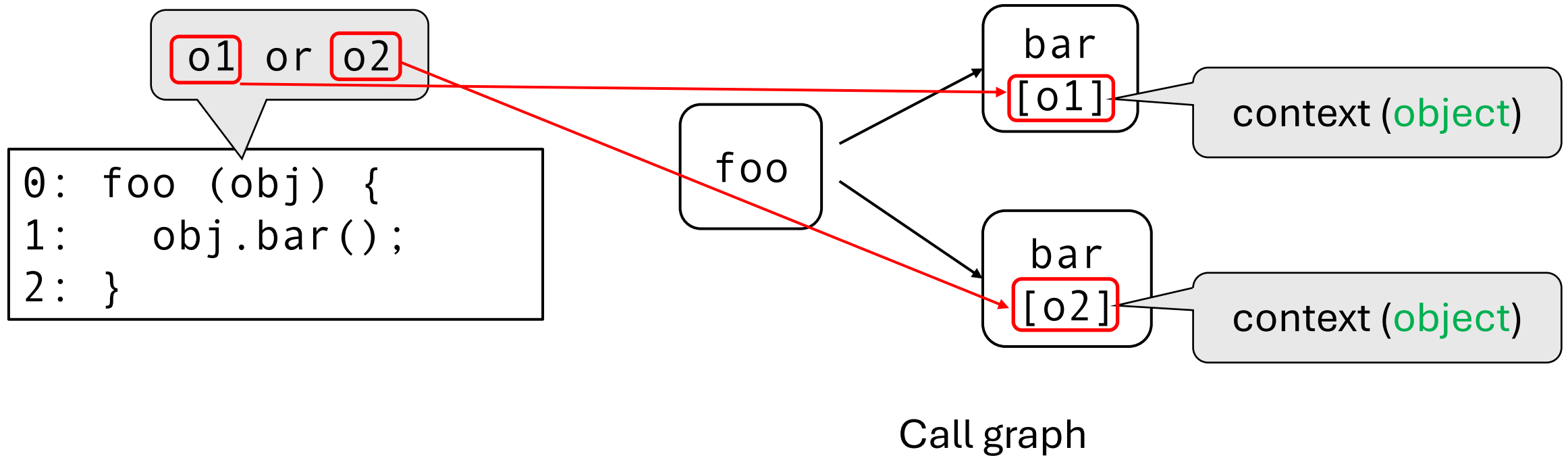
Object Sensitivity considers “what”



Call graph

# Call-Site Sensitivity vs Object Sensitivity

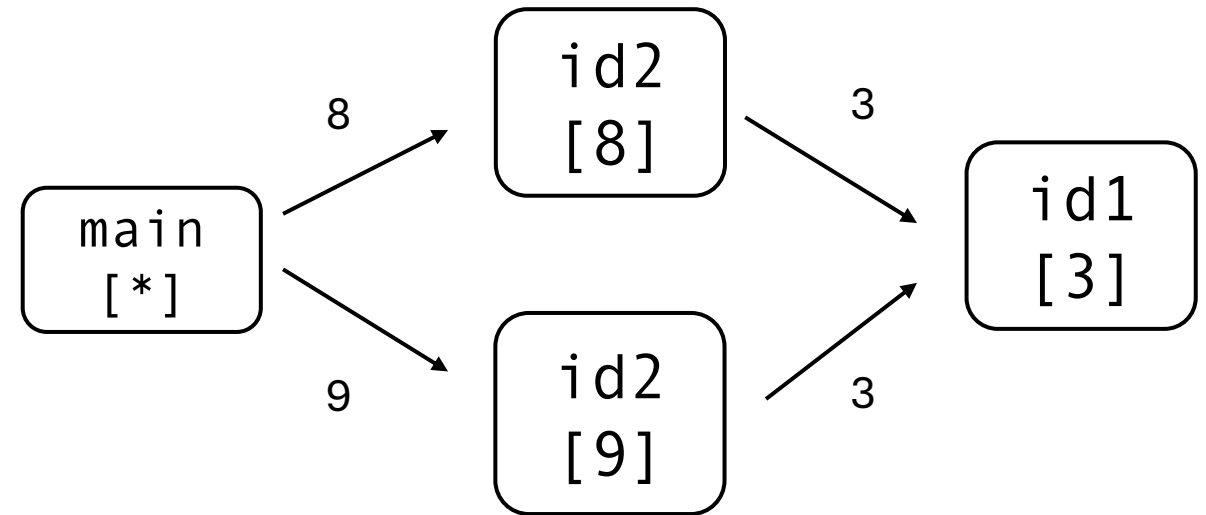
Object Sensitivity considers “what”



# Call-Site Sensitivity vs Object Sensitivity

Example: **weakness** of **call-site sensitivity** and **strength** of **object sensitivity**

```
1 : class C {  
2 :   id1(v) { return v };  
3 :   id2(v) { return id1(v) };  
4 : }  
5 : main() {  
6 :   c1 = new C();  
7 :   c2 = new C();  
8 :   A a = (A) c1.id2(new A());  
9 :   B b = (B) c2.id2(new B());  
10: }
```

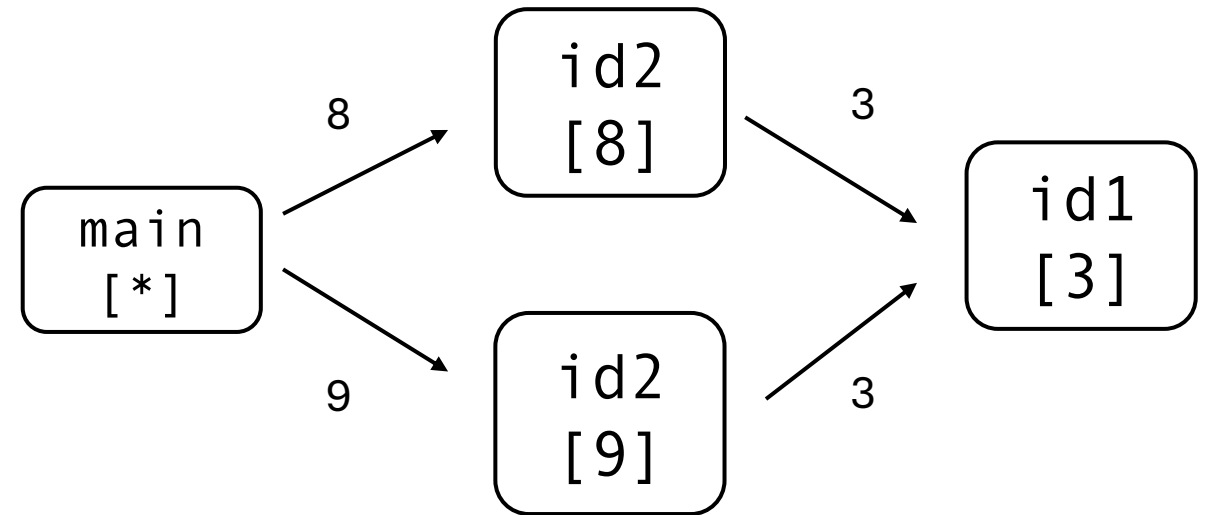


Call graph of 1-call-site

# Call-Site Sensitivity vs Object Sensitivity

Example: **weakness** of **call-site sensitivity** and **strength** of **object sensitivity**

```
1 : class C {  
2 :   id1(v) { return v };  
3 :   id2(v) { return id1(v) };  
4 : }  
5 : main() {  
6 :   c1 = new C();  
7 :   c2 = new C();  
8 :   A a = (A) c1.id2(new A());  
9 :   B b = (B) c2.id2(new B());  
10: }
```



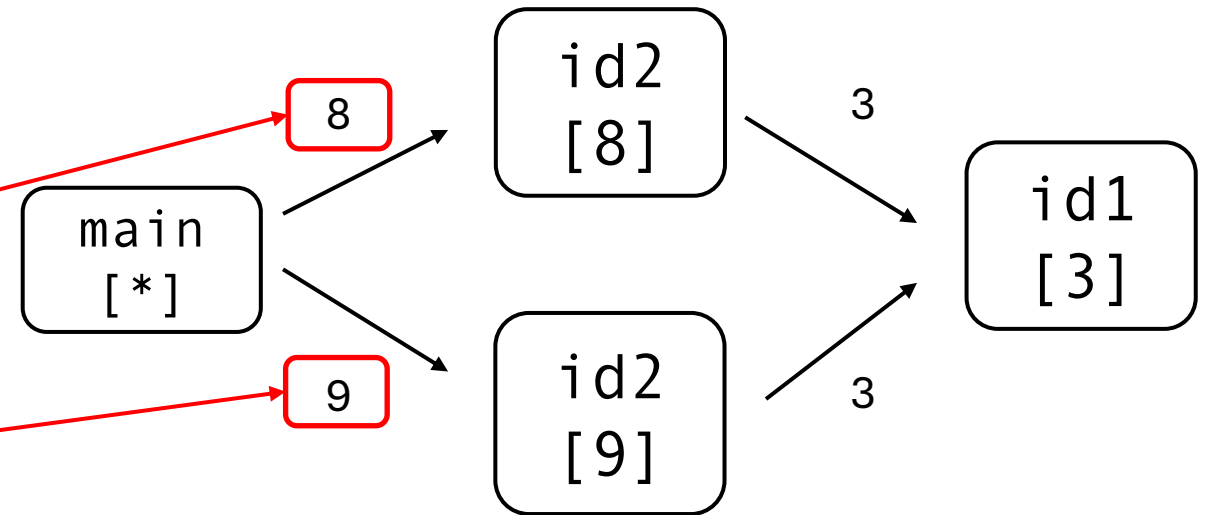
Call graph of 1-call-site



# Call-Site Sensitivity vs Object Sensitivity

Example: **weakness** of **call-site sensitivity** and **strength** of **object sensitivity**

```
1 : class C {  
2 :   id1(v) { return v };  
3 :   id2(v) { return id1(v) };  
4 : }  
5 : main() {  
6 :   c1 = new C();  
7 :   c2 = new C();  
8 :   A a = (A) c1.id2(new A());  
9 :   B b = (B) c2.id2(new B());  
10: }
```



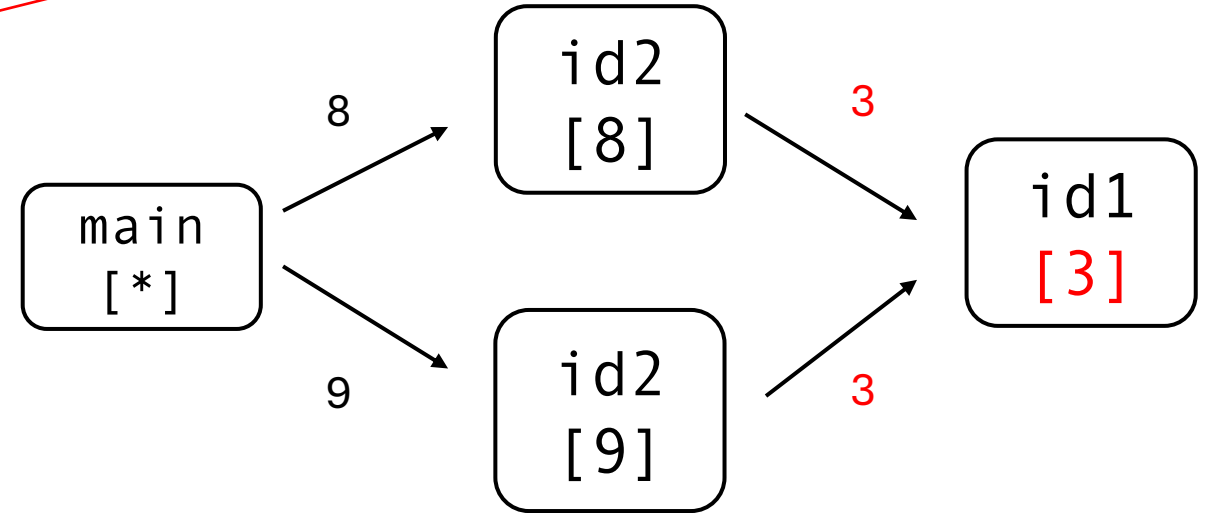
Call graph of 1-call-site

# Call-Site Sensitivity vs Object Sensitivity

Example: weakness of call-site sensitivity and

Weakness:  
nested method calls

```
1 : class C {  
2 :   id1(v) { return v };  
3 :   id2(v) { return id1(v) };  
4 : }  
5 : main() {  
6 :   c1 = new C();  
7 :   c2 = new C();  
8 :   A a = (A) c1.id2(new A());  
9 :   B b = (B) c2.id2(new B());  
10: }
```

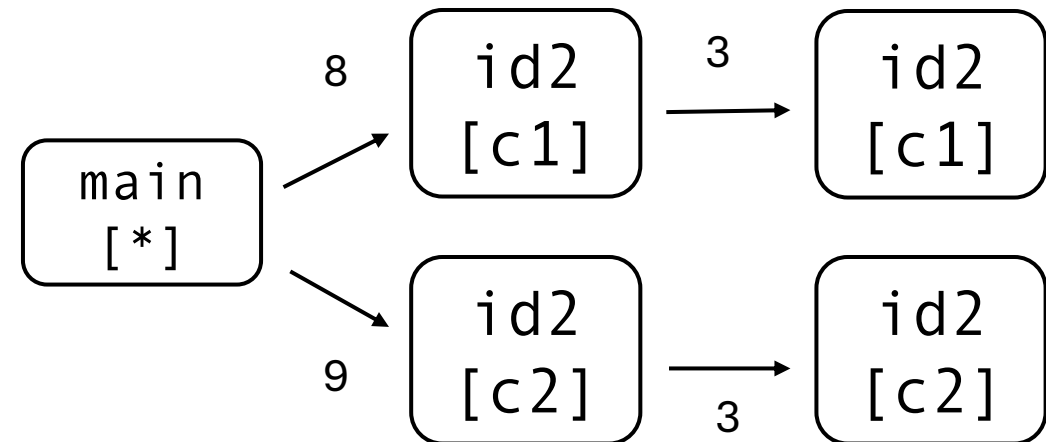


Call graph of 1-call-site

# Call-Site Sensitivity vs Object Sensitivity

Example: weakness of call-site sensitivity and strength of object sensitivity

```
1 : class C {  
2 :     id1(v) { return v };  
3 :     id2(v) { return id1(v) };  
4 : }  
5 : main() {  
6 :     c1 = new C();  
7 :     c2 = new C();  
8 :     A a = (A) c1.id2(new A());  
9 :     B b = (B) c2.id2(new B());  
10: }
```



Call graph of 1-object

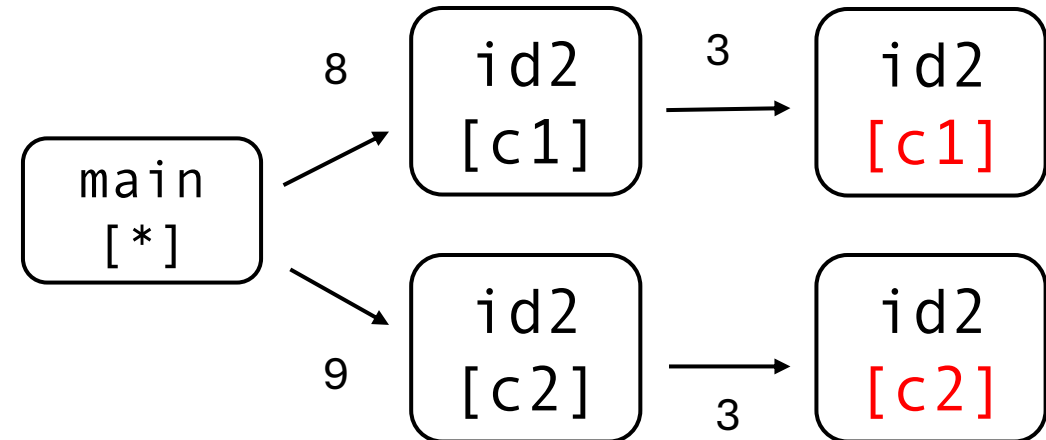
# Call-Site Sensitivity vs Object Sensitivity

Example: weak

Strength:  
c1 or c2

ty and strength of object sensitivity

```
1 : class C {  
2 :   id1(v) { return v };  
3 :   id2(v) { return id1(v) };  
4 : }  
5 : main() {  
6 :   c1 = new C();  
7 :   c2 = new C();  
8 :   A a = (A) c1.id2(new A());  
9 :   B b = (B) c2.id2(new B());  
10: }
```

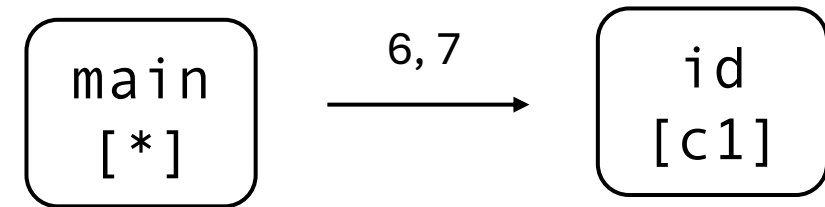


Call graph of 1-object

# Call-Site Sensitivity vs Object Sensitivity

Example: **weakness** of **object sensitivity** and **strength** of **call-site sensitivity**

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

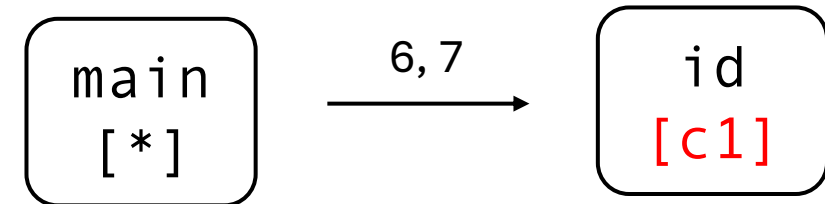


Call graph of 1-object

# Call-Site Sensitivity vs Object Sensitivity

Example: **weakness** of **object sensitivity** and **strength** of **call-site sensitivity**

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



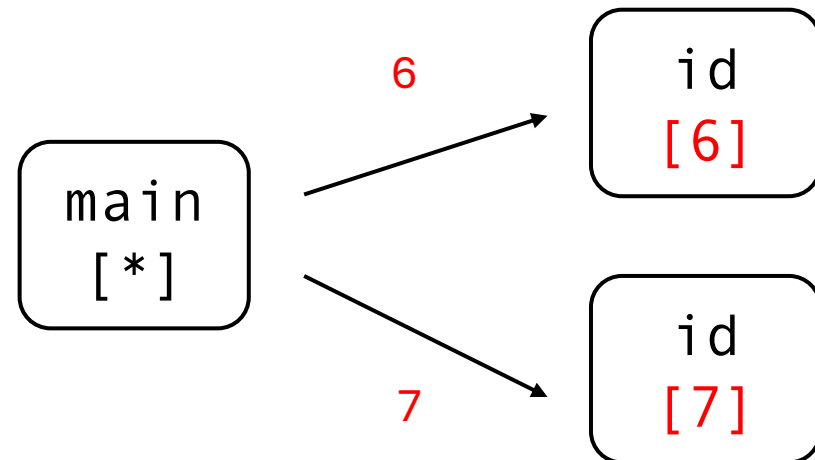
Call graph of 1-object

**Weakness:**  
The two method calls share the same object c1

# Call-Site Sensitivity vs Object Sensitivity

Example: weakness of object sensitivity and strength of call-site sensitivity

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



Call graph of 1-call-site

Strength:

Easily separates the two method calls

# Call-Site Sensitivity vs Object Sensitivity

The Status Quo: Object sensitivity outperforms call-site sensitivity

1981: Call-Site sensitivity proposed

2002: Object sensitivity proposed

2002 ~ 2010: Object vs Call-site

2010 ~ 2022: Object win

- Researchers focused on improving Object sensitivity approach



# Call-Site Sensitivity vs Object Sensitivity

The Status Quo: Object sensitivity outperforms call-site sensitivity

1981: Call-Site sensitivity proposed

2002: Object sensitivity proposed

2002 ~ 2010: Object vs Call-site

2018: Context tunneling proposed

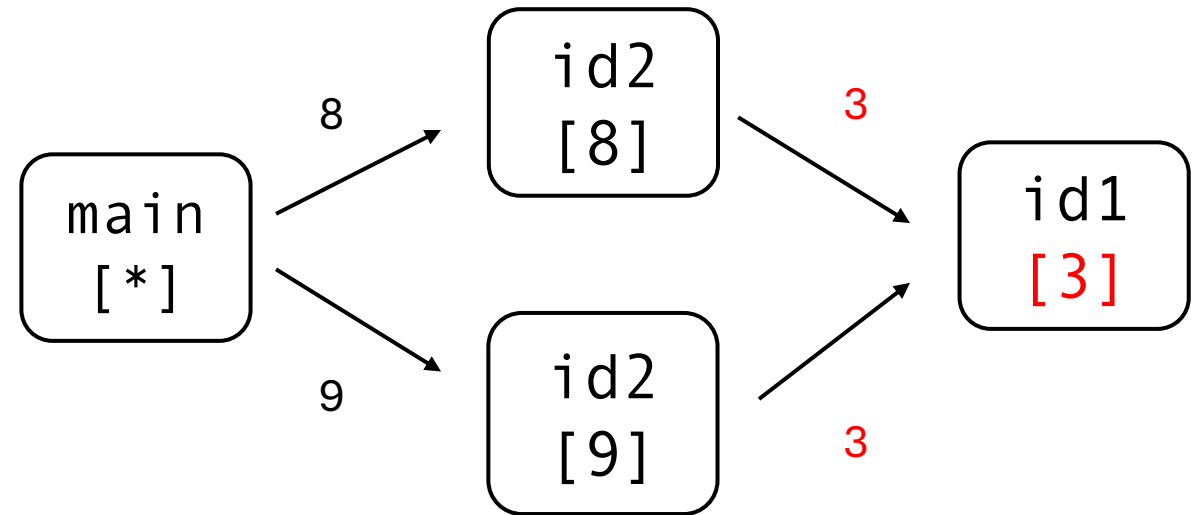
2010 ~ 2022: Object win

- Researchers focused on improving Object sensitivity approach

# Call-Site Sensitivity vs Object Sensitivity

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

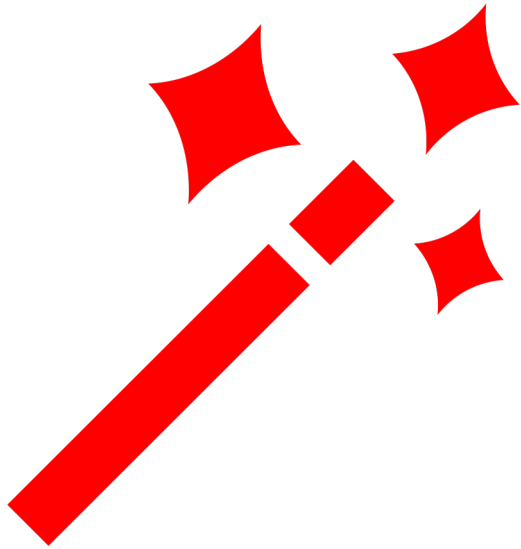
```
1 : class C {  
2 :   id1(v) { return v };  
3 :   id2(v) { return id1(v) };  
4 : }  
5 : main() {  
6 :   c1 = new C();  
7 :   c2 = new C();  
8 :   A a = (A) c1.id2(new A());  
9 :   B b = (B) c2.id2(new B());  
10: }
```



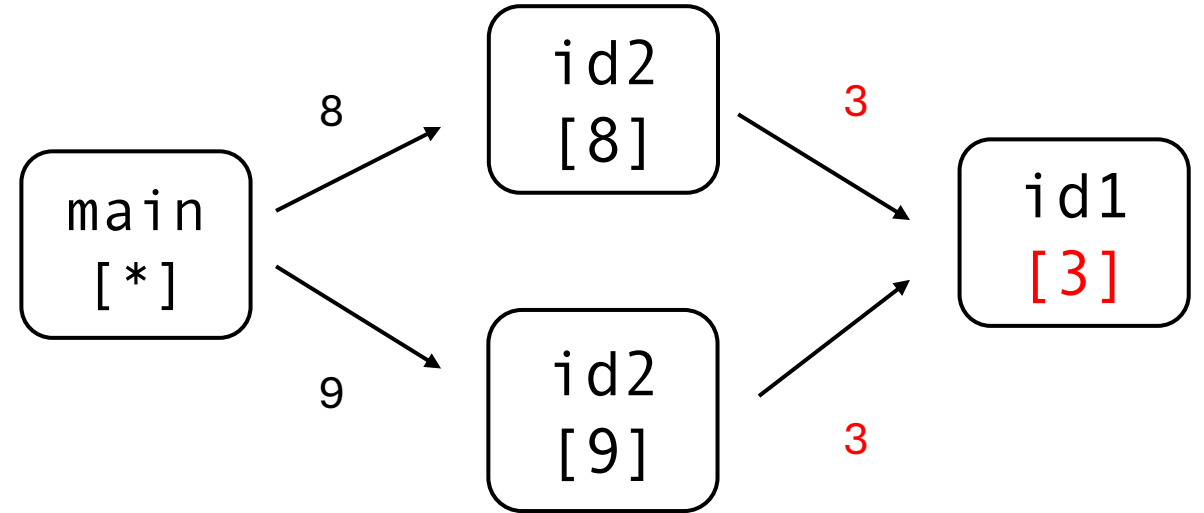
Call graph of 1-call-site

# Call-Site Sensitivity vs Object Sensitivity

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



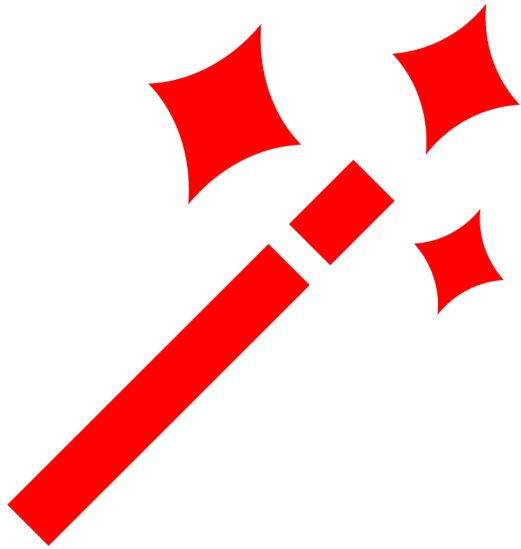
**Context Tunneling**



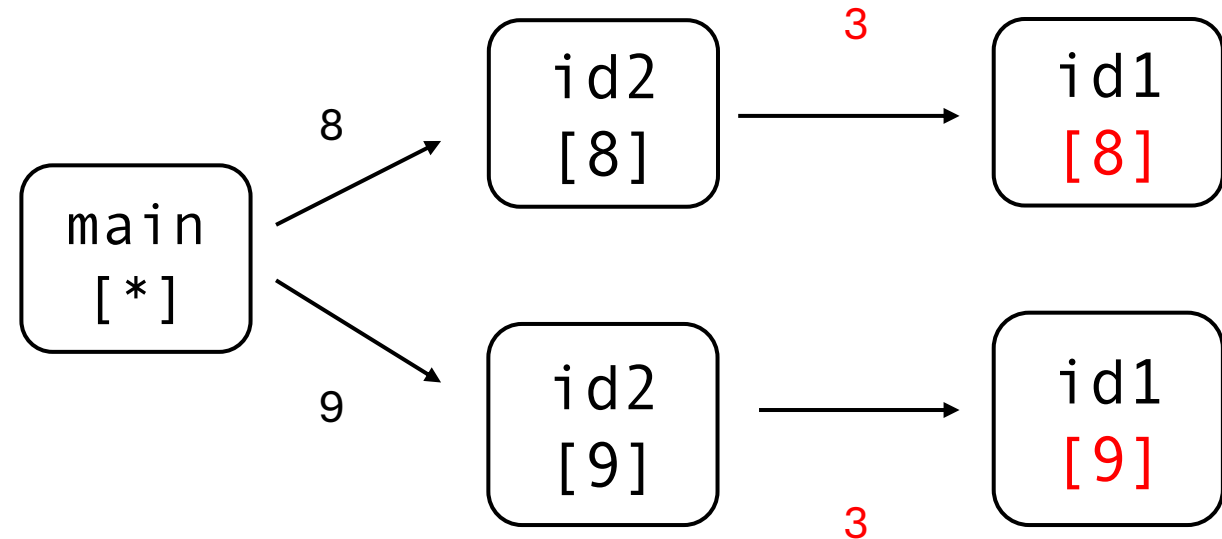
Call graph of 1-call-site

# Call-Site Sensitivity vs Object Sensitivity

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



**Context Tunneling**



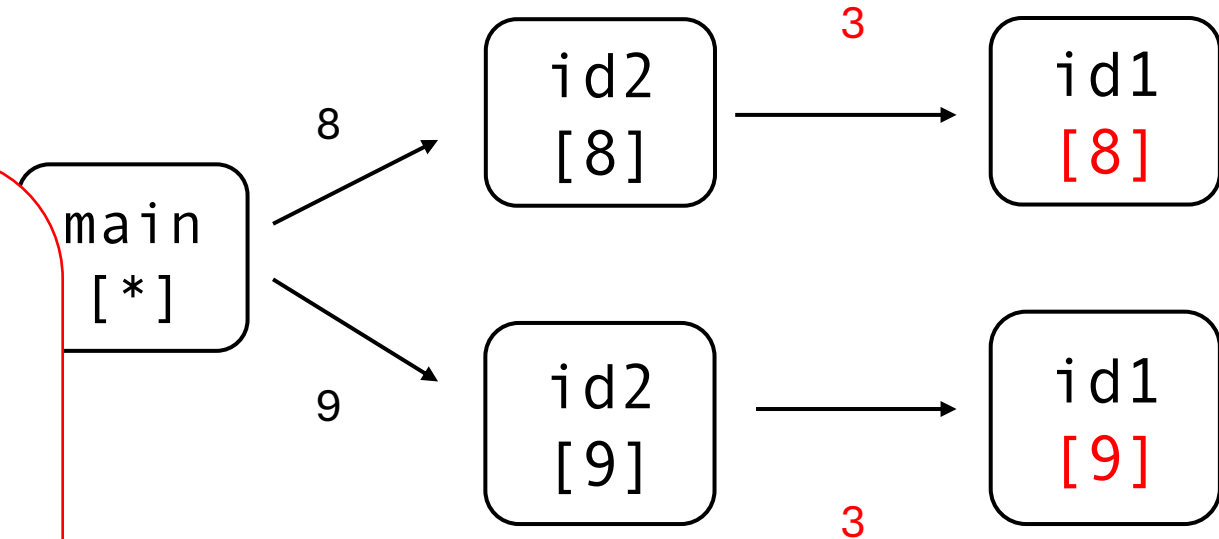
Call graph of 1-call-site  
with **context tunneling**  
( $T = \{3\}$ )

# Call-Site Sensitivity vs Object Sensitivity

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

## Tunneling abstraction:

- Determines where to apply context tunneling



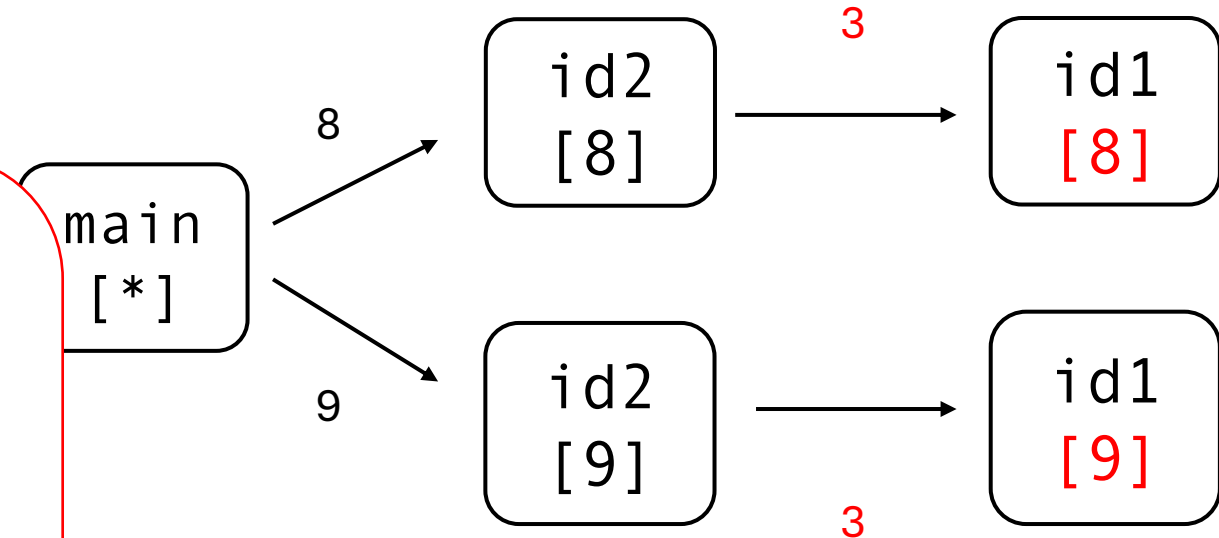
Call graph of 1-call-site  
with context tunneling  
(**T = {3}**)

# Call-Site Sensitivity vs Object Sensitivity

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

## Tunneling abstraction:

- Determines where to apply context tunneling
- Refers a set of unimportant call-sites that should not be used as a context elements



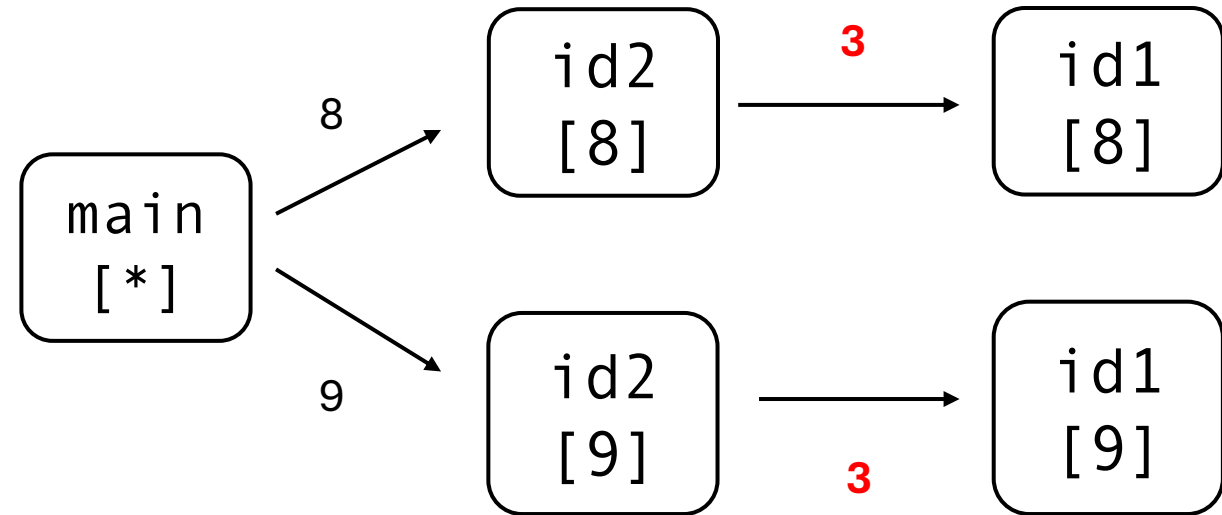
Call graph of 1-call-site  
with context tunneling  
(**T = {3}**)

# Call-Site Sensitivity vs Object Sensitivity

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

**Apply context tunneling:**

1. Detect tunneling abstraction (**T = {3}**)



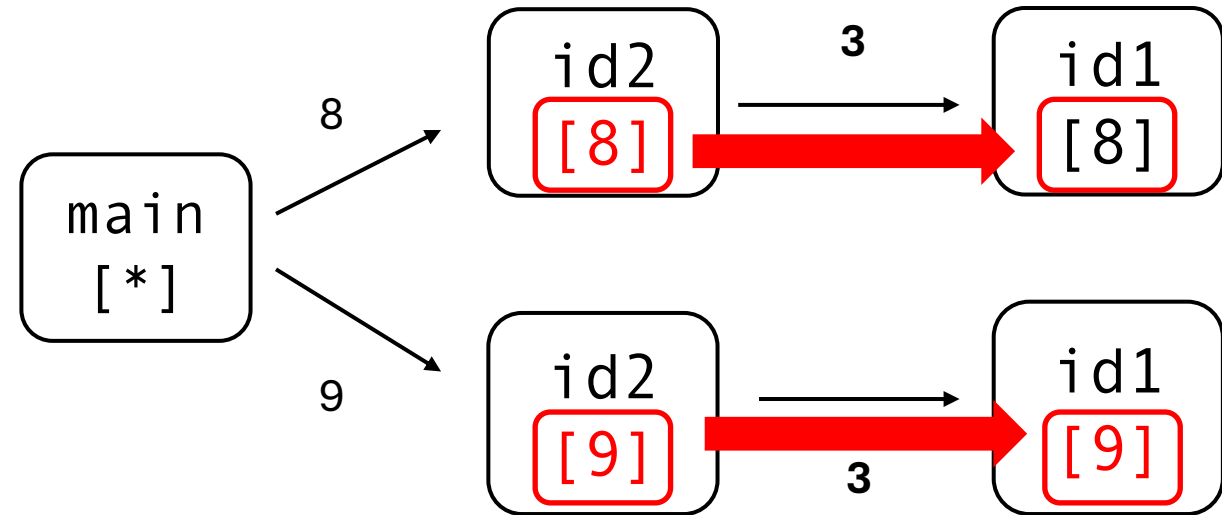
Call graph of 1-call-site  
with context tunneling  
(**T = {3}**)

# Call-Site Sensitivity vs Object Sensitivity

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

## Apply context tunneling:

1. Detect tunneling abstraction ( $T = \{3\}$ )
2. Inherit caller method's context

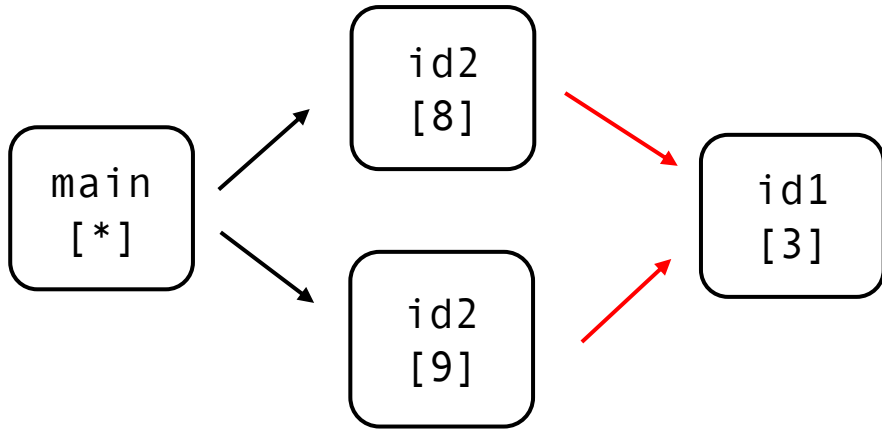


Call graph of 1-call-site  
with context tunneling  
( $T = \{3\}$ )

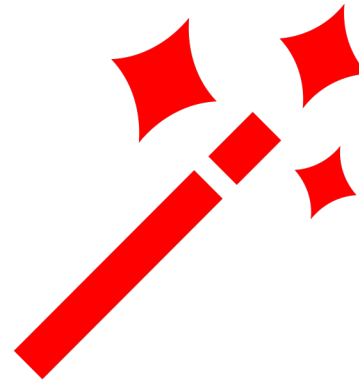


# Call-Site Sensitivity vs Object Sensitivity

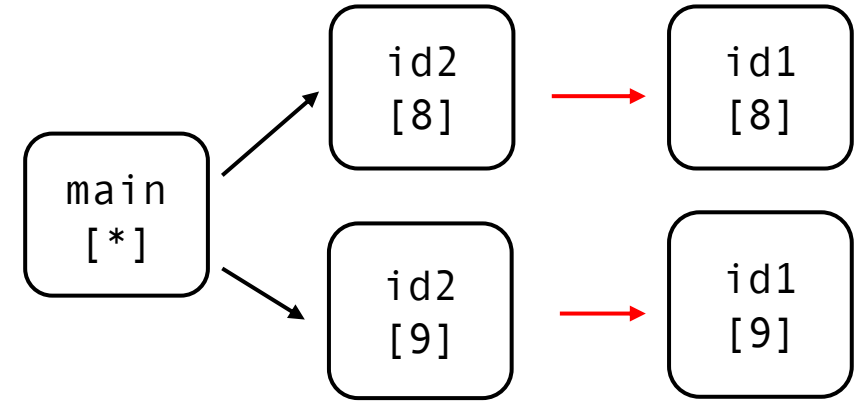
Context tunneling can remove the **weakness** of **call-site sensitivity**



Call graph of 1-call-site



Context Tunneling



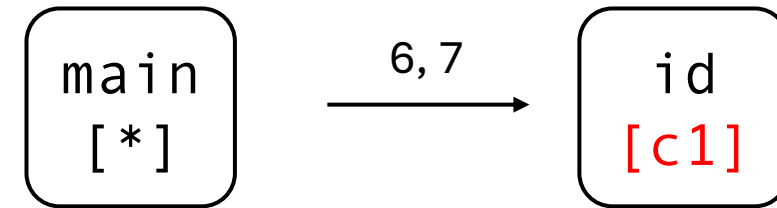
Call graph of 1-call-site  
with **context tunneling**  
( $T = \{3\}$ )

# Call-Site Sensitivity vs Object Sensitivity

Context tunneling **can't** remove the **weakness** of **object sensitivity**

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

**Weakness:**  
The two method calls share the same object c1

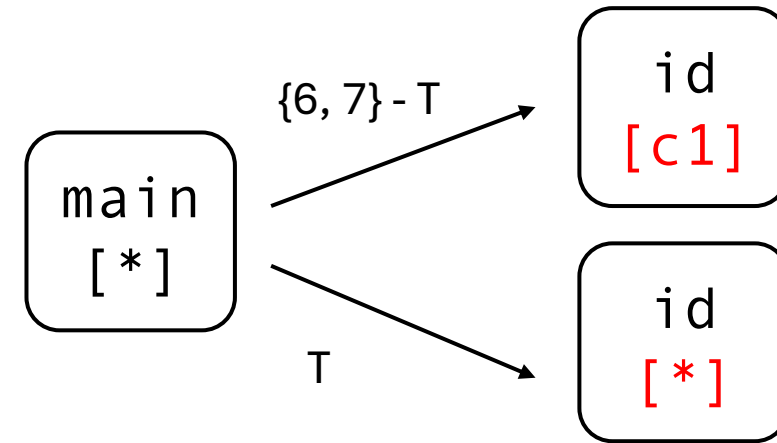


Call graph of 1-object

# Call-Site Sensitivity vs Object Sensitivity

Context tunneling **can't** remove the **weakness** of **object sensitivity**

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



Call graph of 1-call-site  
with context tunneling  
(**T = {?}**)

Unable to separate the two method calls  
with **two context (c1 and \*)**

# Call-Site Sensitivity vs Object Sensitivity

## Obervation

When context tunneling is included,

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

# Call-Site Sensitivity vs Object Sensitivity

## Observation

When context tunneling is included,

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

## Claim

When context tunneling is included,

- **Call-site sensitivity** is **more precise** than **object sensitivity**

# Call-Site Sensitivity vs Object Sensitivity

## Observation

When context tunneling is included,

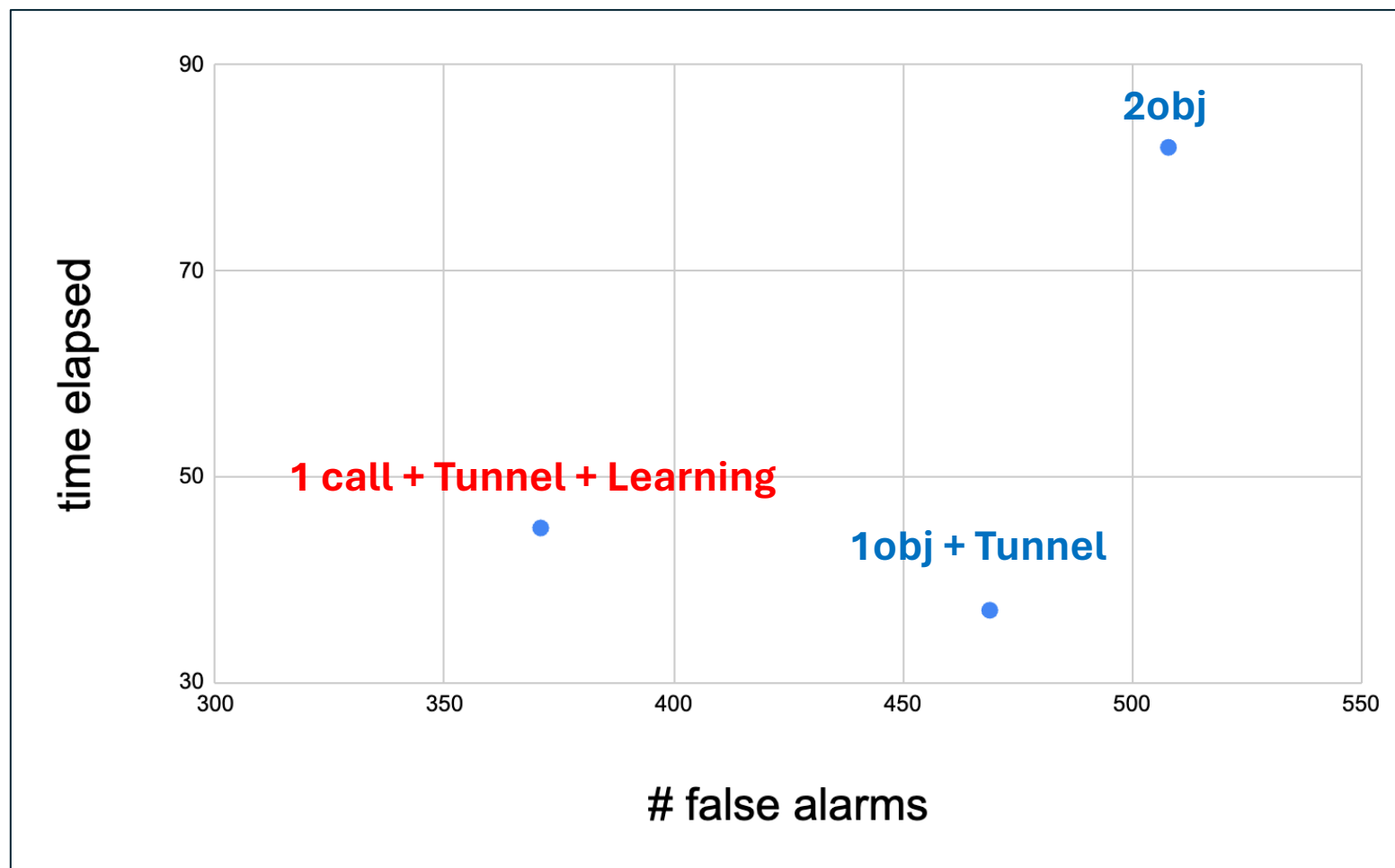
- Weakness of **call-site sensitivity** is removed
- Weakness of **object sensitivity** is not removed

## Claim

When context tunneling is included,

- **Call-site sensitivity** is **more precise** than **object sensitivity**

# Results



# More details about **Obj2CFA**

- Obj2CFA uses **simulation** technique
  1. Runs the object-sensitive analysis to obtain its **call graph**
  2. Infers a **tunneling abstraction**
- **Simulation**: Finding more precise call-site sensitivity, but expensive
  - Apply Simulation-guided **Learning** to Improve scalability



# Obj2CFA - Simulation

**Simulation** takes a call-graph and infers a tunneling abstraction

```
1 : class C {  
2 :     id1(v) { return v };  
3 :     id2(v) { return id1(v)};  
4 :     foo() {  
5 :         A a = (A)this.id1(new A());  
6 :         B b = (B)this.id1(new B());}  
7 : }  
8 : main() {  
9 :     c1 = new C();  
10:    c2 = new C();  
11:    c3 = new C();  
12:    A a = (A) c1.id2(new A());  
13:    B b = (B) c2.id2(new B());  
14:    c3.foo();  
15: }
```

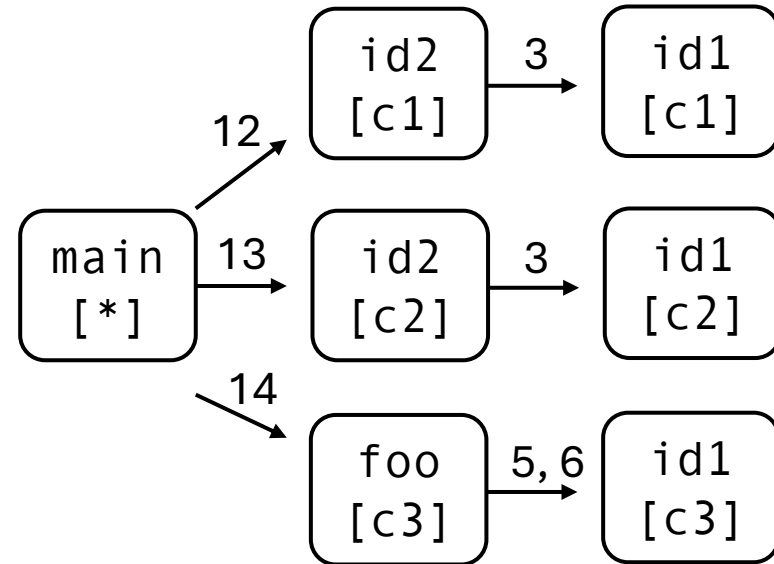
**Weakness of call-site sensitivity**

**Weakness of object sensitivity**

# Obj2CFA - Simulation

**Simulation** takes a call-graph and infers a tunneling abstraction

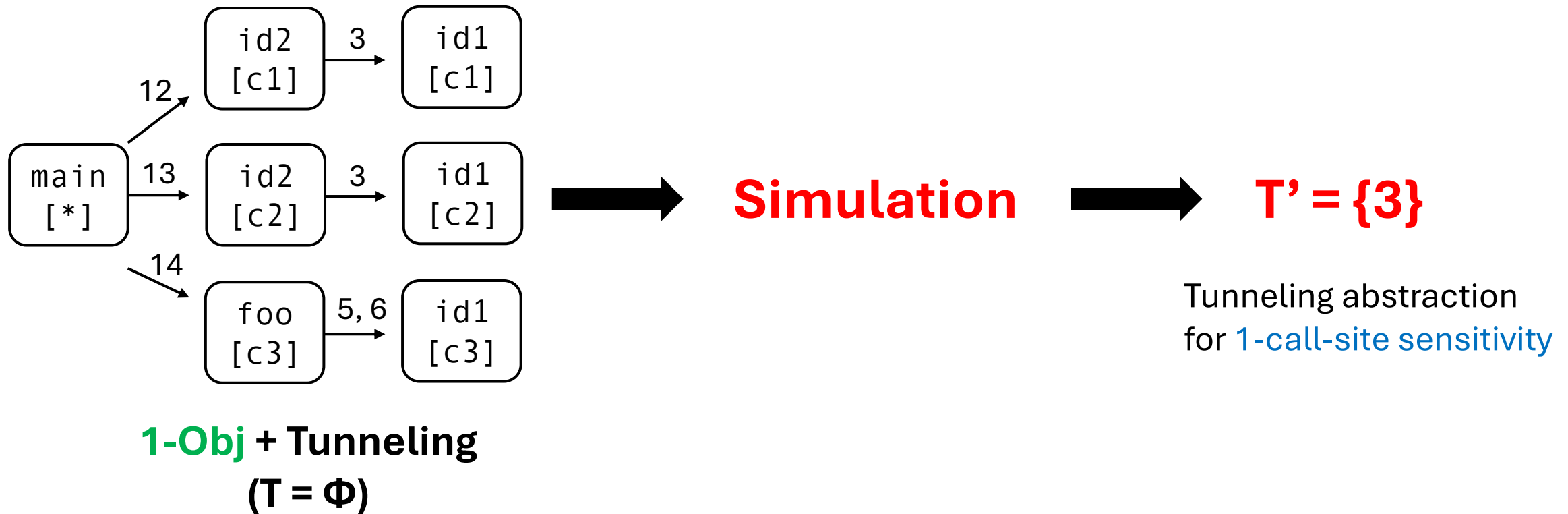
```
1 : class C {
2 :   id1(v) { return v };
3 :   id2(v) { return id1(v)};
4 :   foo() {
5 :     A a = (A)this.id1(new A());
6 :     B b = (B)this.id1(new B());}
7 : }
8 : main() {
9 :   c1 = new C();
10:   c2 = new C();
11:   c3 = new C();
12:   A a = (A) c1.id2(new A());
13:   B b = (B) c2.id2(new B());
14:   c3.foo();
15: }
```



**1-Obj + Tunneling**  
**( $T = \Phi$ )**

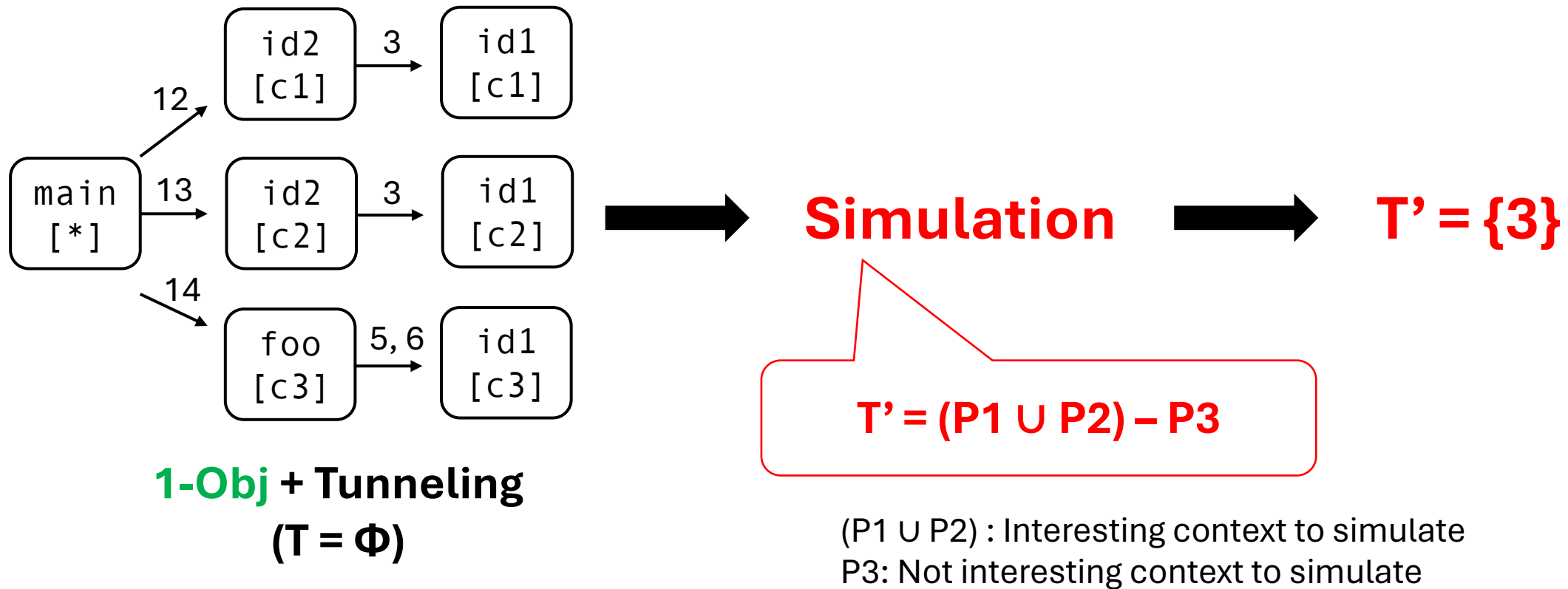
# Obj2CFA - **Simulation**

**Simulation** takes a call-graph and infers a tunneling abstraction



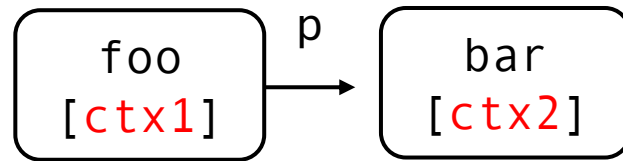
# Obj2CFA - **Simulation**

**Simulation** takes a call-graph and infers a tunneling abstraction



# Obj2CFA - **Simulation**

**Simulation** takes a call-graph and infers a tunneling abstraction

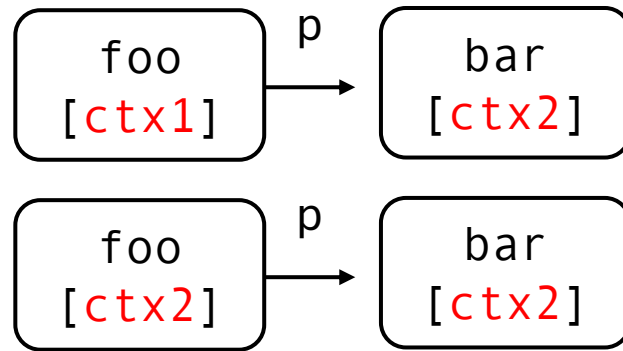


$$T' = (P1 \cup P2) - P3$$

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

# Obj2CFA - **Simulation**

**Simulation** takes a call-graph and infers a tunneling abstraction

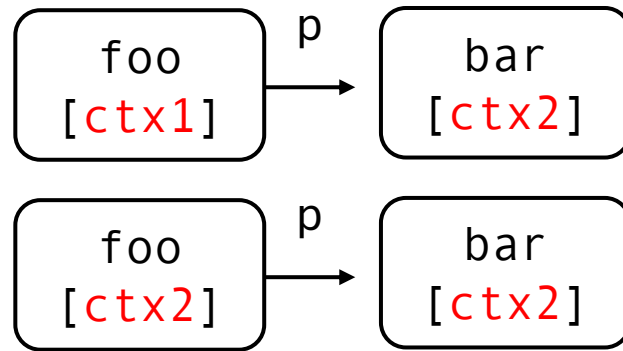


$$T' = (P1 \cup P2) - P3$$

- Property 1: caller and callee methods have the **same context**
- Property 2: **different caller** contexts imply **different callee** contexts

# Obj2CFA - Simulation

**Simulation** takes a call-graph and infers a tunneling abstraction

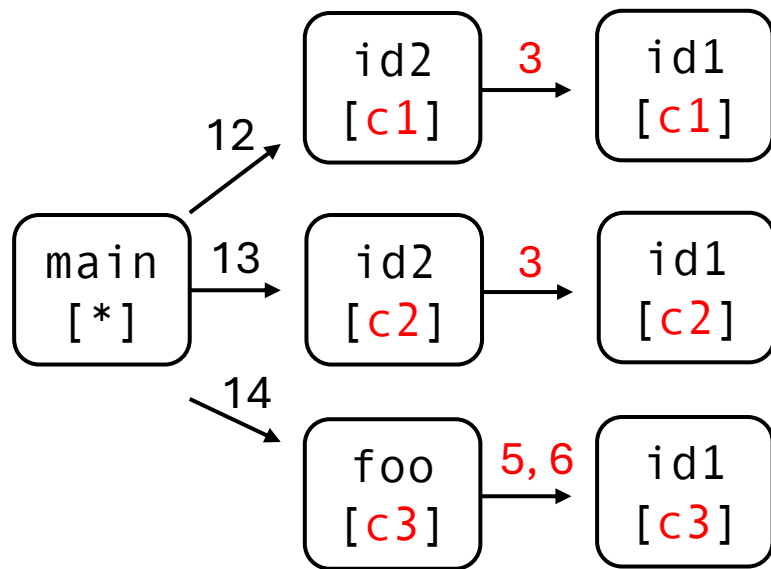


$$T' = (P1 \cup P2) - P3$$

- Property 1: caller and callee methods have the **same context**
- Property 2: **different caller** contexts imply **different callee** contexts
- Property 3: given object sensitivity **produced only one context**

# Obj2CFA - Simulation

**Simulation** takes a call-graph and infers a tunneling abstraction



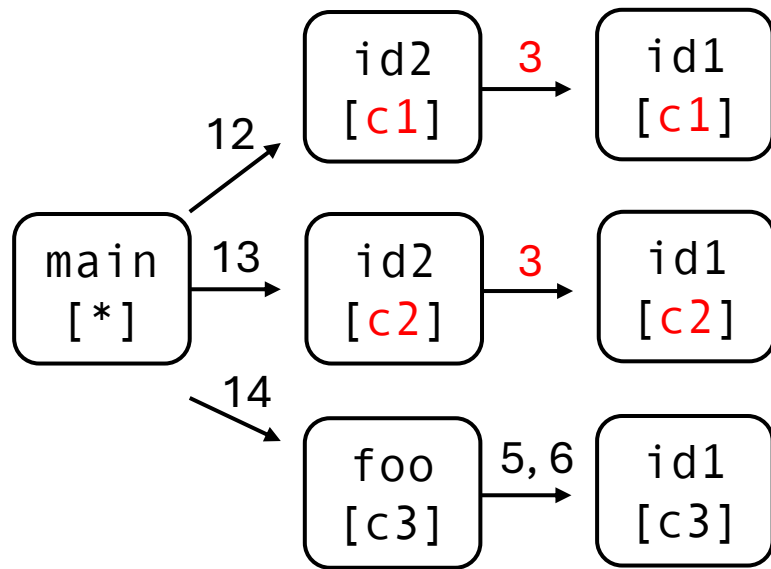
- P1: caller and calle methods have the same context  
P1 = {3, 5, 6}

**1-Obj + Tunneling**  
(T =  $\Phi$ )



# Obj2CFA - Simulation

**Simulation** takes a call-graph and infers a tunneling abstraction

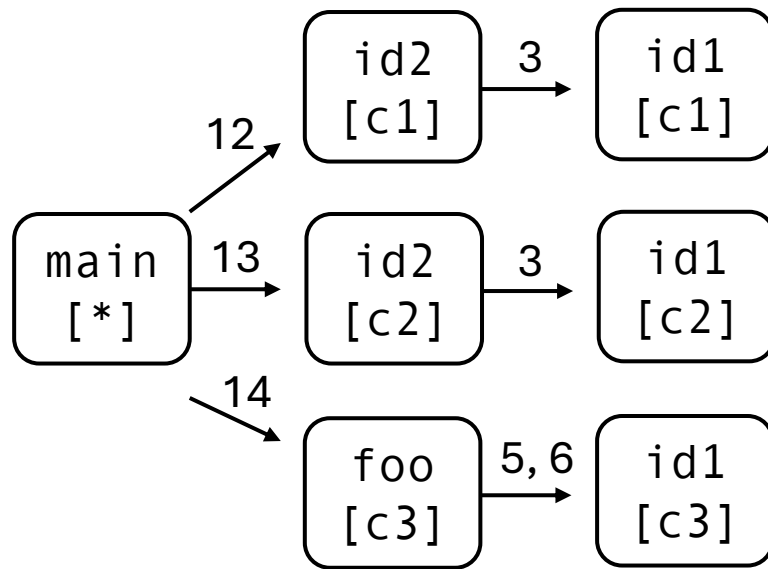


**1-Obj + Tunneling**  
( $T = \Phi$ )

- P1: caller and callee methods have the same context  
 $P1 = \{3, 5, 6\}$
- P2: different caller context imply different callee context  
 $P2 = \{3\}$

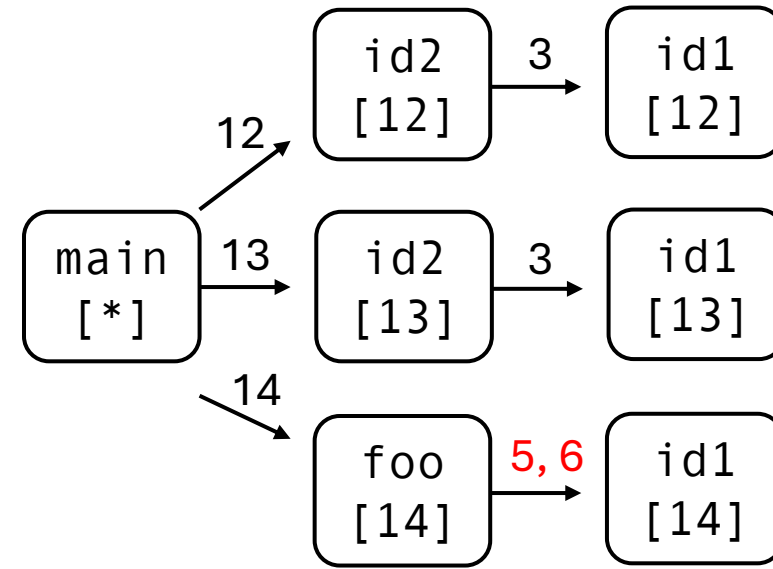
# Obj2CFA - **Simulation**

**Simulation** takes a call-graph and infers a tunneling abstraction



**1-Obj + Tunneling**  
( $T = \Phi$ )

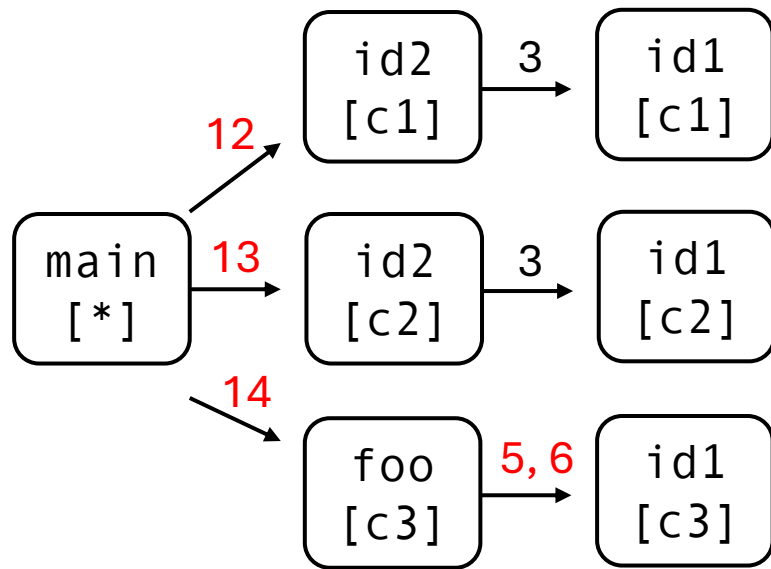
**=**



**1-call-site + Tunneling**  
( $T = P1 \cup P2 = \{3, 5, 6\}$ )

# Obj2CFA - Simulation

**Simulation** takes a call-graph and infers a tunneling abstraction

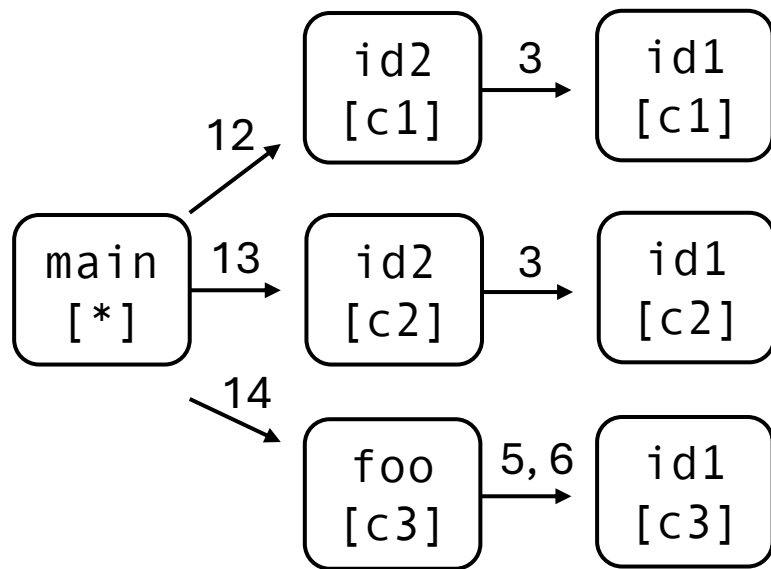


**1-Obj + Tunneling**  
( $T = \Phi$ )

- P1: caller and calle methods have the same context  
 $P1 = \{3, 5, 6\}$
- P2: different caller context imply different callee context  
 $P2 = \{3\}$
- P3: given object sensitivity produced only one context  
 $P3 = \{5, 6, 12, 13, 14\}$

# Obj2CFA - Simulation

**Simulation** takes a call-graph and infers a tunneling abstraction



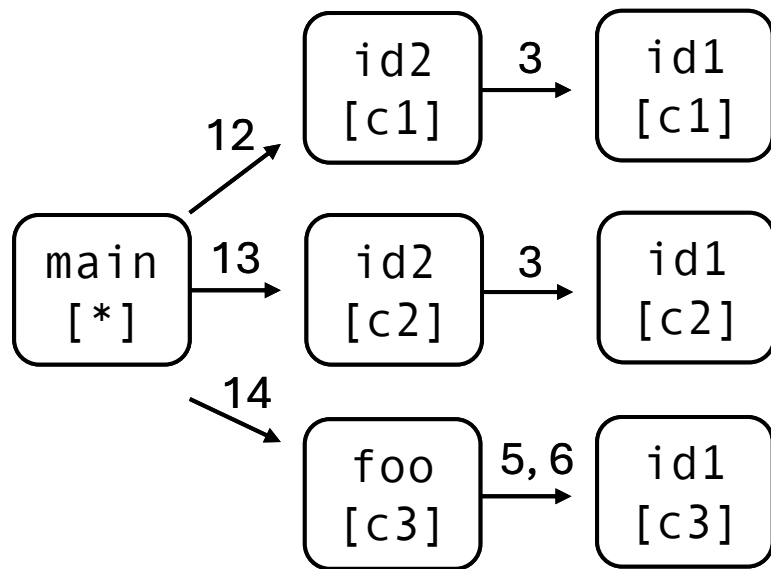
**1-Obj + Tunneling**  
( $T = \Phi$ )

- P1: caller and callee methods have the same context  
 $P1 = \{3, 5, 6\}$
- P2: different caller context imply different callee context  
 $P2 = \{3\}$
- P3: given object sensitivity produced only one context  
 $P3 = \{5, 6, 12, 13, 14\}$

$$T' = (P1 \cup P2) - P3 = \{3\}$$

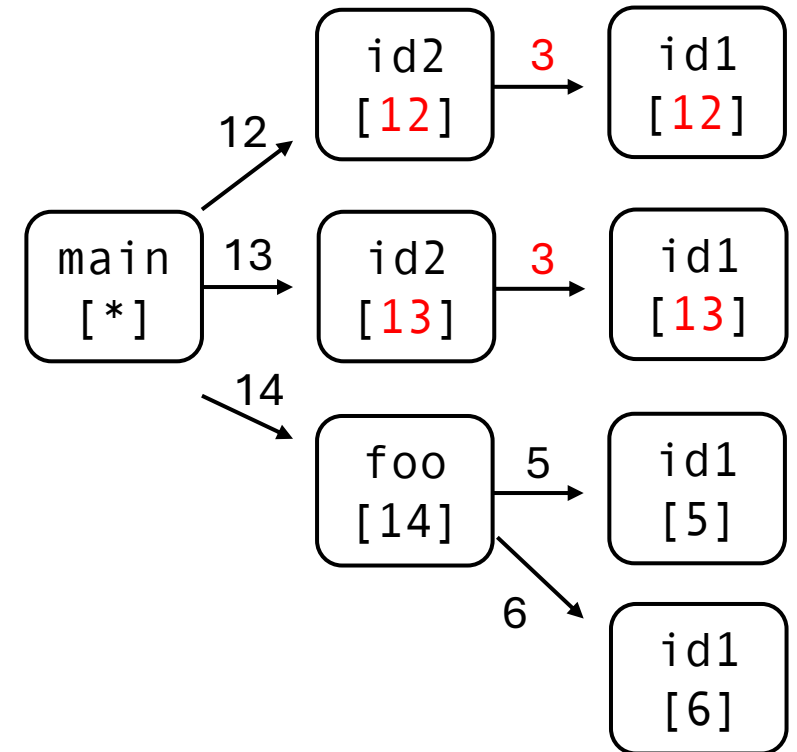
# Obj2CFA - **Simulation**

**Simulation** takes a call-graph and infers a tunneling abstraction



**1-Obj + Tunneling**  
( $T = \Phi$ )

→ **Simulation** →



**1-call-site + Tunneling**  
( $T' = \{3\}$ )

# Obj2CFA – Simulation-guided Learning

- Simulation is expensive
  - simulation requires object sensitivity to obtain call graph
- Learning
  - Given training programs and simulated tunneling abstractions,
  - Find a model that produces similar tunneling abstractions
- The authors modified the features of the context tunneling paper [1] to capture the specific simulated behavior

# Evaluation

- **Doop**
  - Pointer analysis framework for JAVA
- Dataset
  - 12 Java programs (10 DaCapo 2006 programs + 2 real-world open source programs)
- Baselines
  - 1-object-sensitive with tunneling [1]
  - 1-call-site sensitivity with tunneling [1]
  - 2-object-sensitive [2]
  - 2-object sensitivity [3]

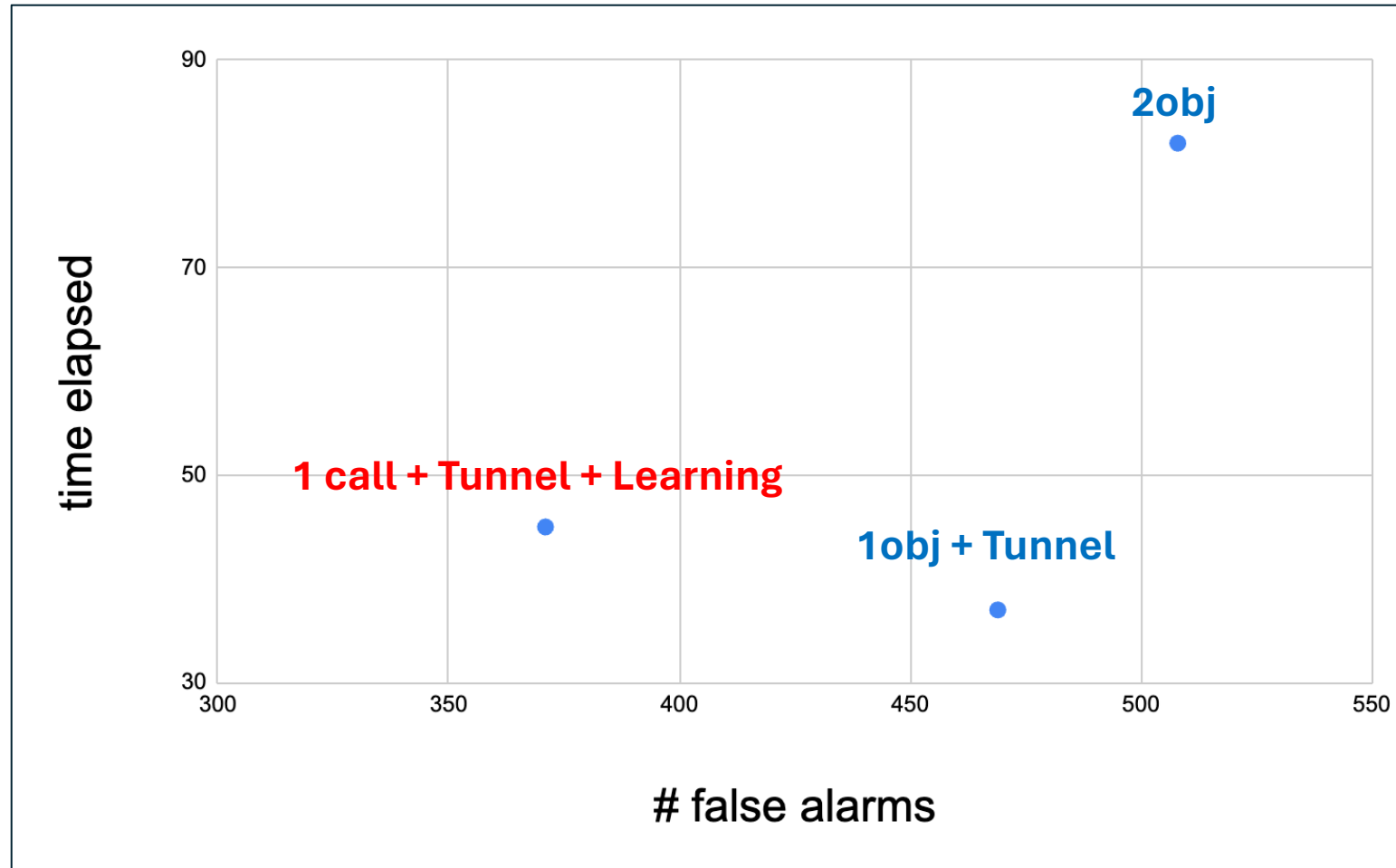
[1] Jeong et al. 2017. Data-driven Context-sensitivity for Points-to Analysis. Proc. ACM Program. Lang. 1, OOPSLA

[2] Smaragdakis et al. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation

[3] Li et al. 2018a. Precision-guided Context Sensitivity for Pointer Analysis. Proc. ACM Program. Lang. 2, OOPSLA

# Results

Obj2CFA outperformed the baselines in terms of **precision** and **scalability** (time elapsed)





# Summary

- Until recently, **call-site sensitivity** is known as a worse context than **object sensitivity**
- However **context tunneling** can make **call-site sensitivity** analysis to a good context flavor
- **Cotext tunneling** can transform **object sensitivity** to **call-site sensitivity**
- **Cotext tunneling** can improve the precision of **call-site sensitivity**

# Appendix: Call-Site Sensitivity vs Object Sensitivity

## Object Sensitivity wins!

### Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java

Ana Milanova    Atanas Rountev    Barbara G. Ryder  
Department of Computer Science  
Rutgers University  
New Brunswick, NJ 08901  
(milanova, rountev, ryder)@cs.rutgers.edu

#### ABSTRACT

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. Improving the precision of practical points-to analysis is important because points-to information has a wide variety of client applications in optimizing compilers and software engineering tools. In this paper 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 objects on which this method is 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. This information is needed for various compiler optimizations and software engineering tools. We present a new form of side-effect analysis for Java which is based on object-sensitive points-to analysis.

We have implemented one instantiation of our parameterized object-sensitive points-to analysis. We compare this instantiation with a context-insensitive points-to analysis for Java which is based on Andersen's analysis for C [4]. On a set of 23 Java programs, our experiments show that the two analyses have comparable cost. In some cases the object-sensitive analysis is actually faster than the context-insensitive analysis. Our results also show that object sensitivity significantly improves the precision of side-effect analysis, call graph construction, and virtual call resolution. These experiments demonstrate that object-sensitive analyses can achieve significantly better precision than context-insensitive ones, while at the same time remaining efficient and practical.

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.  
1574-7421/02/10-22:24, 2002, Rome, Italy  
Copyright 2002 ACM 1-58113-562-9/02/0007...\$5.00

#### 1. INTRODUCTION

*Points-to analysis* is a fundamental static analysis used by optimizing Java compilers and software engineering tools to determine the set of objects whose addresses may be stored in reference variables and reference object fields. These *points-to sets* are typically computed by constructing one or more *points-to graphs*, which serve as abstractions of the run-time memory states of the analyzed program. (An example of a points-to graph is shown in Figure 1, which is discussed in Section 2.1)

Optimizing Java compilers can use points-to information to perform various optimizations such as virtual call resolution, removal of unnecessary synchronization, and stack-based object allocation. *Points-to analysis* is also a prerequisite for a variety of other analyses—for example, *side-effect analysis*, which determines the memory locations that may be modified by the execution of a statement, and *def-use analysis*, which identifies pairs of statements that set the value of a memory location and subsequently use that value. These analyses are necessary to perform compiler optimizations such as code motion and partial redundancy elimination. In addition, such analyses are needed in the context of software engineering tools: for example, *def-use analysis* is needed for program slicing and data-flow-based testing. *Points-to analysis* is a crucial prerequisite for employing these analyses and optimizations.

Because of this wide range of applications, it is important to investigate approaches for precise and efficient computation of points-to information. The two major dimensions in the design space of points-to analysis are flow sensitivity and context sensitivity. Intuitively, *flow-sensitive analyses* take into account the flow of control between program points inside a method, and compute separate solutions for these points. *Flow-insensitive analyses* ignore the flow of control between program points, and therefore can be less precise and more efficient than flow-sensitive analyses. *Context-sensitive analyses* distinguish between the different contexts under which a method is invoked, and analyze the method separately for each context. *Context-insensitive analyses* do not separate the different invocation contexts for a method, which improves efficiency at the expense of some possible precision loss.

Recent work [19, 26, 15, 20] has shown that flow- and context-insensitive points-to analysis for Java can be efficient and practical even for large programs, and therefore

### 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 Lhotak and Hendren's PtoSet, 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 PtoSet: 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 PtoSet and Whaley et al.'s bdbdbdb, directly addressing open problems in past literature. Finally, our implementation is modular and can be easily configured to analyze 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 analyses, 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



### 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 Batrox 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, Batrox 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 2005], program verification [Fink et al. 2008; Pradel et al. 2012] and program understanding [Li et al. 2016; Sridharan

\*Corresponding author

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, xxm@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@umaz.gr.



This work is licensed under a Creative Commons Attribution 4.0 International License.  
© 2021 Copyright held by the owner(s)/author(s).  
2475-1421/2021/10-ART147  
<https://doi.org/10.1145/3485524>

Proc. ACM Program. Lang., Vol. 5, No. OOPSLA, Article 147, Publication date: October 2021.

- [1] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (January 2005), 1–41. <https://doi.org/10.1145/1044834.1044835>
- [2] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. *SIGPLAN Not.* 44, 10 (October 2009), 243–262. <https://doi.org/10.1145/1639949.1640108>
- [3] 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>