A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis

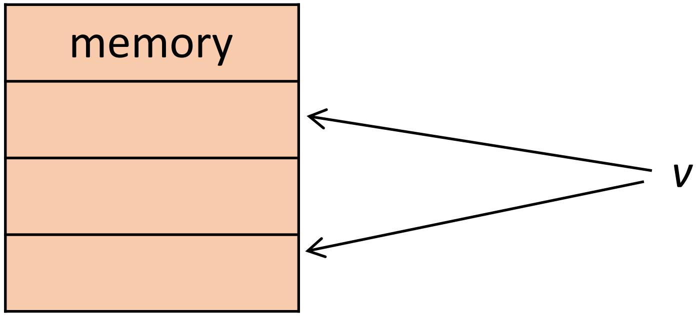
Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao,
Changwei Zou, Yulei Sui, Jingling Xue

School of Computer Science and Engineering,
University of New South Wales, Australia

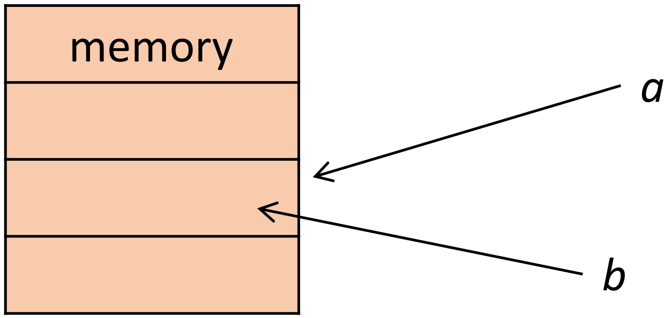
October 26, 2023

Pointer Analysis

- ❑ Programs (in C/C++/Java, …) are full of **pointers** or **references**
- ❑ Answer the following two problems



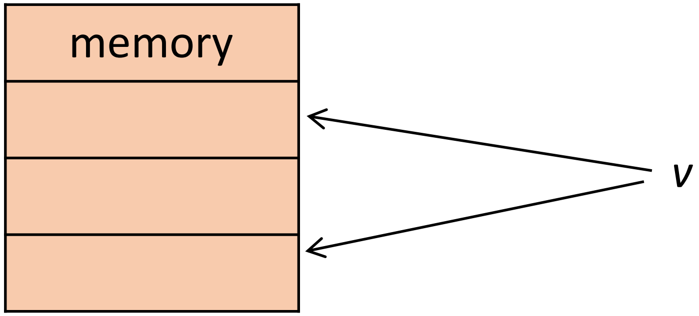
(1) What can a pointer point to?



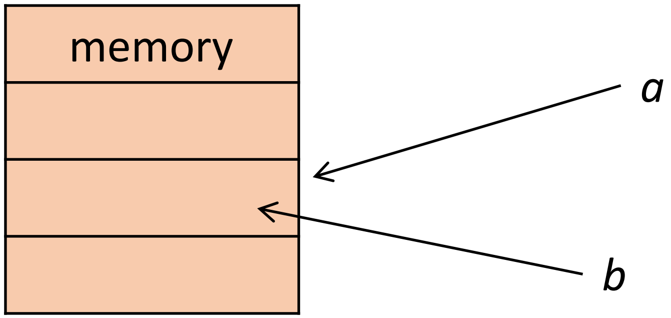
(2) Can *a* and *b* be aliases?

Pointer Analysis

- Programs (in C/C++/Java, …) are full of **pointers** or **references**
- Answer the following two problems



(1) What can a pointer point to?



(2) Can *a* and *b* be aliases?

- Foundation of many Static Program Analysis

Compiler Optimization | Call-graph Construction | Program Verification | Program Understanding | Bug Detection | …

- Implemented in many popular frameworks

Qilin Soot WALA (T. J. WATSON LIBRARIES FOR ANALYSIS) DOOP SVF

Object Sensitive Pointer Analysis (kOBJ)

❑An effective technique to improve the precision

❑Context elements are receiver objects

  - $[o_1, o_2, \cdots, o_k]$

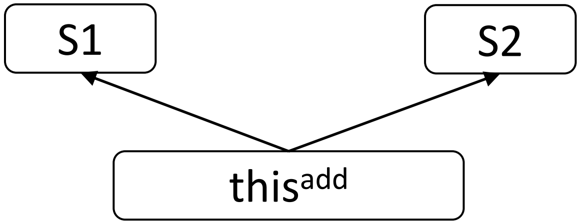
Object Sensitive Pointer Analysis (kOBJ)

- An effective technique to improve the precision
- Context elements are receiver objects
    - $[o_1, o_2, \cdots, o_k]$

- An example
    - two contexts of **HashSet.add** **:** [S1] and [S2].

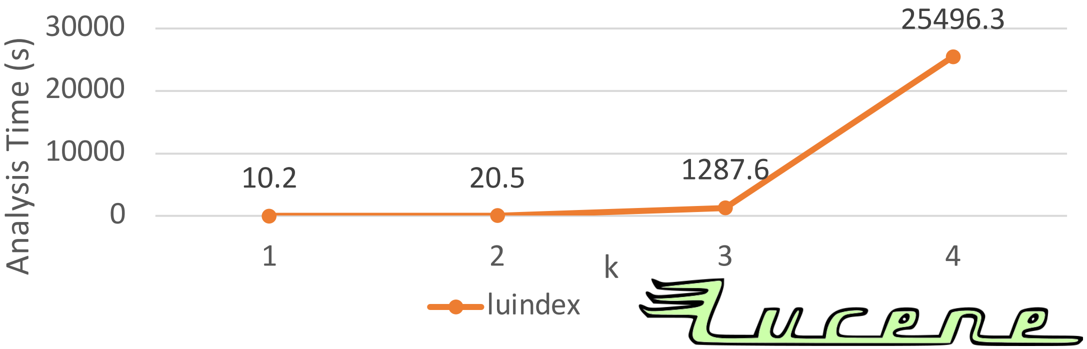
```
HashSet s1 = new HashSet();// S1
HashSet s2 = new HashSet();// S2
s1.add(new Object()); // O1
s2.add(new Object()); // O2
```

In **main()**, analyzed under empty context []



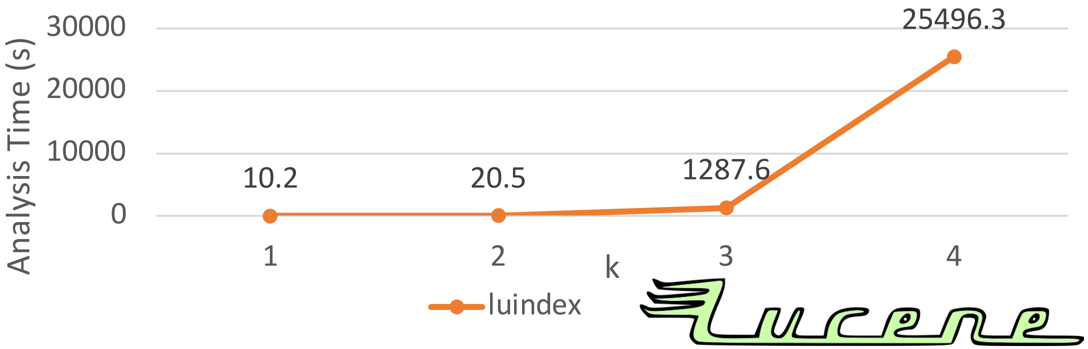
Exponential Explosion Issue

- Increasing k makes kOBJ less Efficient or even Unscalable



Exponential Explosion Issue

❑ Increasing k makes kOBJ less Efficient or even Unscalable



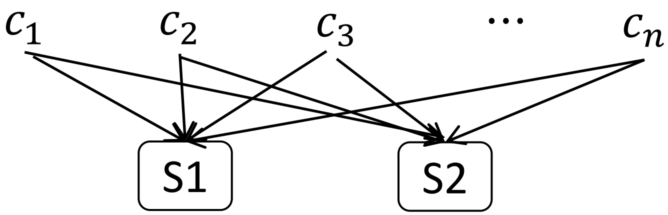
Lucene

❑ Contexts of kOBJ grow exponentially with k

- **HashSet.add :** $2n$ contexts, i.e., [S1, c_i] and [S2, c_i] ($1 \leq i \leq n$).

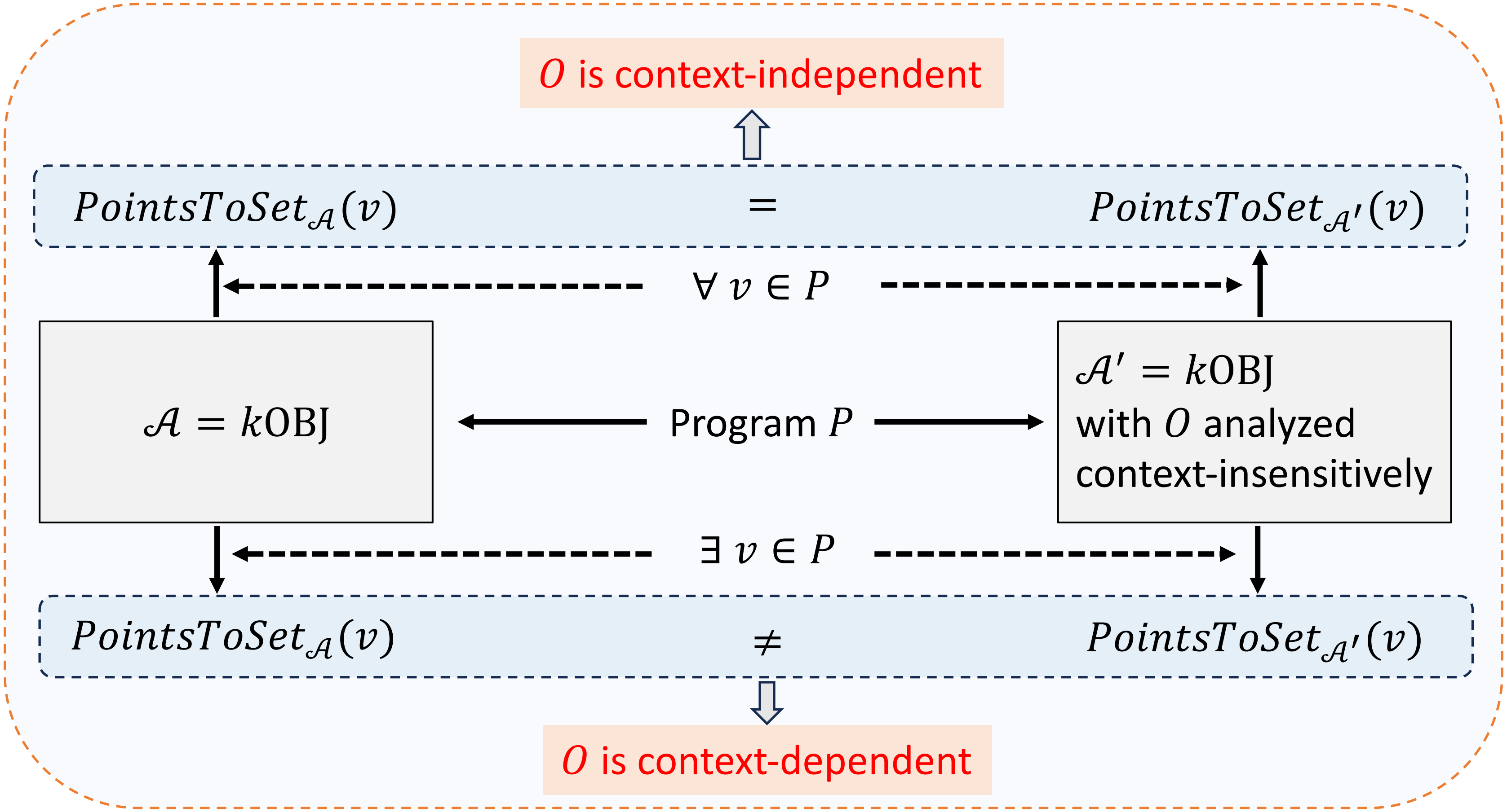
```
HashSet s1 = new HashSet();// S1
HashSet s2 = new HashSet();// S2
s1.add(new Object()); // O1
s2.add(new Object()); // O2
```

analyzed under n different contexts, c_1, $\ldots c_n$



Review: Context Debloating

- ❑ An object (or context element) O is context-dependent if
  - ○ analyzing it context-insensitively in kOBJ will cause some program variables to lose precision.



O is context-independent

$PointsToSet_{\mathcal{A}}(v) = PointsToSet_{\mathcal{A}'}(v)$

$\forall\, v \in P$

$\mathcal{A} = k\text{OBJ}$

Program P

$\mathcal{A}' = k\text{OBJ}$
with O analyzed
context-insensitively

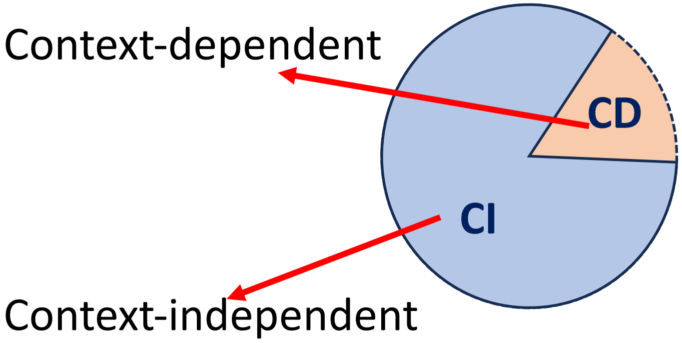
$\exists\, v \in P$

$PointsToSet_{\mathcal{A}}(v) \neq PointsToSet_{\mathcal{A}'}(v)$

O is context-dependent

Review: Context Debloating

- Key Observation:
  - Most **(90%+)** objects are Context-independent
  - E.g., S1 and S2 are locally used and independent of their contexts.
  - Allowing context combinations on them **only** increases analysis time



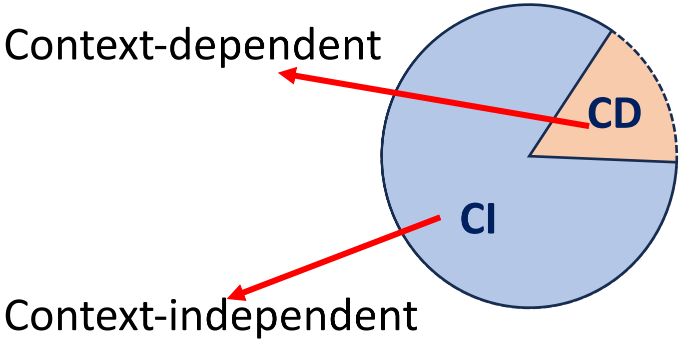
```
HashSet s1 = new HashSet();// S1
HashSet s2 = new HashSet();// S2
s1.add(new Object()); // O1
s2.add(new Object()); // O2
```

analyzed under n contexts, $c_1, \cdots c_n$

Review: Context Debloating

❑ Key Observation:

- **90%+** Context elements (i.e., Objects in kOBJ) are Context-independent
- E.g., S1 and S2 are locally used and independent of their contexts.
- Allowing context combinations on them **only** increases analysis time



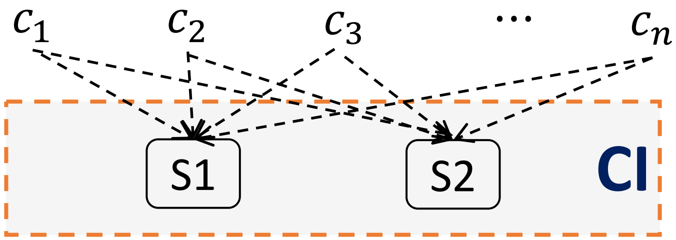
```
HashSet s1 = new HashSet();// S1
HashSet s2 = new HashSet();// S2
s1.add(new Object()); // O1
s2.add(new Object()); // O2
```

analyzed under n contexts, $c_1, \cdots c_n$

❑Key Idea： inhibit context combinations on CI elements

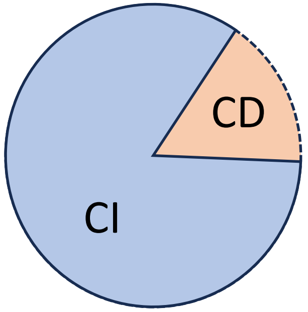
- # contexts of **HashSet.add: $2n$ -> 2**

Precise and Efficient!

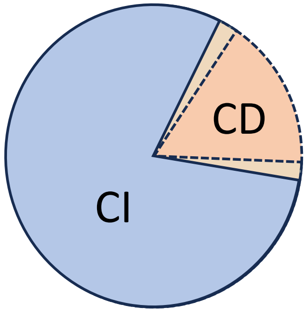


The Challenge in Context Debloating

- ❑How to precisely identify context-(in)dependent objects?

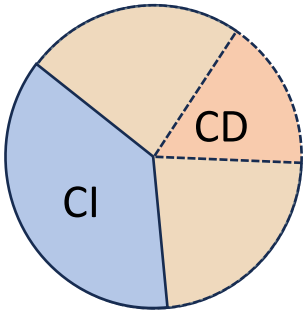


Ideal (Oracle)



Context-free-language(CFL)-based

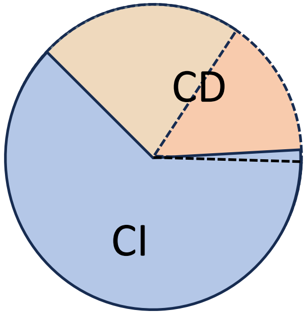
- CFL-based approach is undecidable (OOPSLA’19, TOPLAS’00)
- Eagle, Over-approximation of CFL (OOPSLA’19)
  - selects almost all non-trivial **CI**s as **CD**s
  - Very limited speedups, e.g., 1.5X



Eagle

Context Debloating Approaches

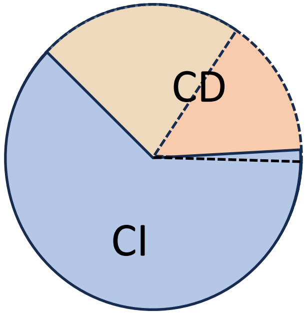
- ☐ **Conch:** approximates CFL with three linearly verifiable conditions
  - ○ field-insensitive (still too conservative)
  - ○ **limits further performance improvement**



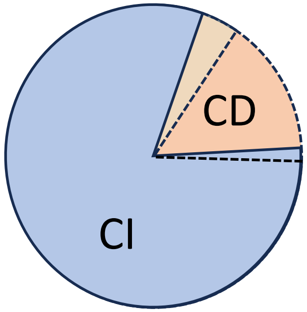
Conch (ASE’21, TOSEM’23)

Context Debloating Approaches

- ❑ **Conch:** approximates CFL with three linearly verifiable conditions
  - field-insensitive (still too conservative)
  - **limits further performance improvement**



Conch (ASE’21, TOSEM’23)



DebloaterX

- ❑ **DebloaterX**
  - 1-limited field-sensitive (e.g, O.f.*)
  - **significant performance improvement with negligible precision loss**

	k = 1	k = 2	k = 3	k = 4
X-kOBJ	10.2s	5.7s	10.3s	14.3s
kOBJ	10.2s	20.5s	1278.6s	25496.3s
C-kOBJ	10.2s	7.6s	363.1s	555.0s

luindex

DebloaterX: Our Approach

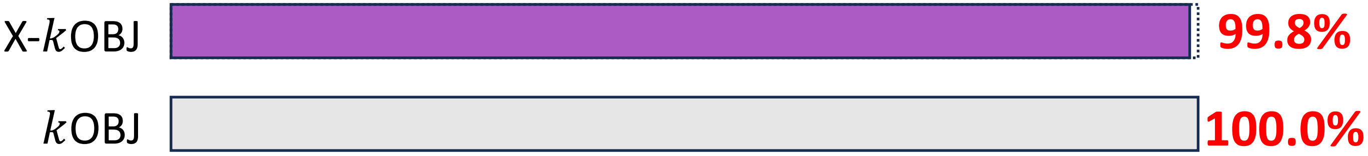
- ❑Identify context-dependent objects by usage patterns
  - ○ We believe Patterns are **finite**!

DebloaterX: Our Approach

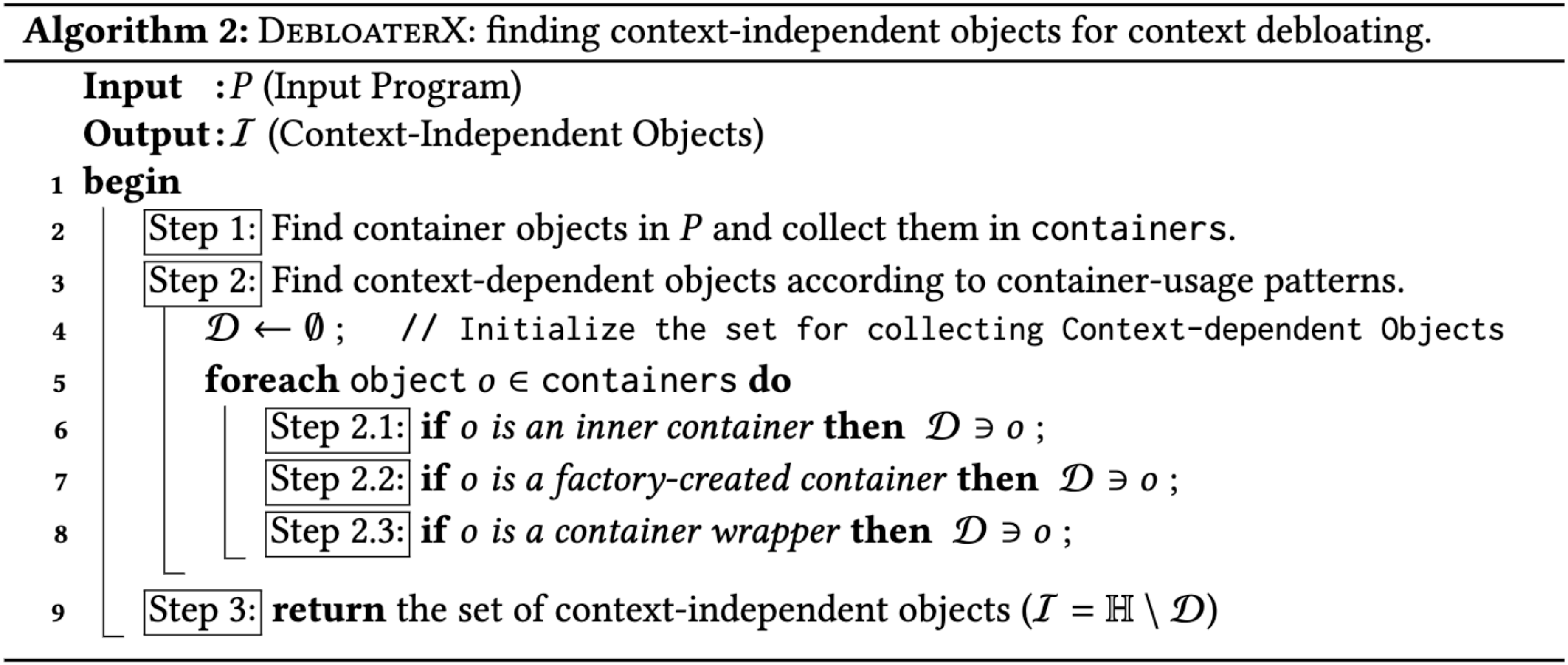
- ❑ Identify context-dependent objects by usage patterns
  - ○ We believe Patterns are **finite**!
- ❑ Three Container-Usage Patterns
  - ○ Inner Containers
  - ○ Factory-Created Containers
  - ○ Container Wrappers

DebloaterX: Our Approach

- Identify context-dependent objects by usage patterns
  - We believe Patterns are **finite**!
- Three Container-Usage Patterns
  - Inner Containers
  - Factory-Created Containers
  - Container Wrappers
- The above **three** Patterns are enough to preserve **99.8%** of precision in real-world programs.



DebloaterX: Overall Algorithm



Algorithm 2: DEBLOATERX: finding context-independent objects for context debloating.

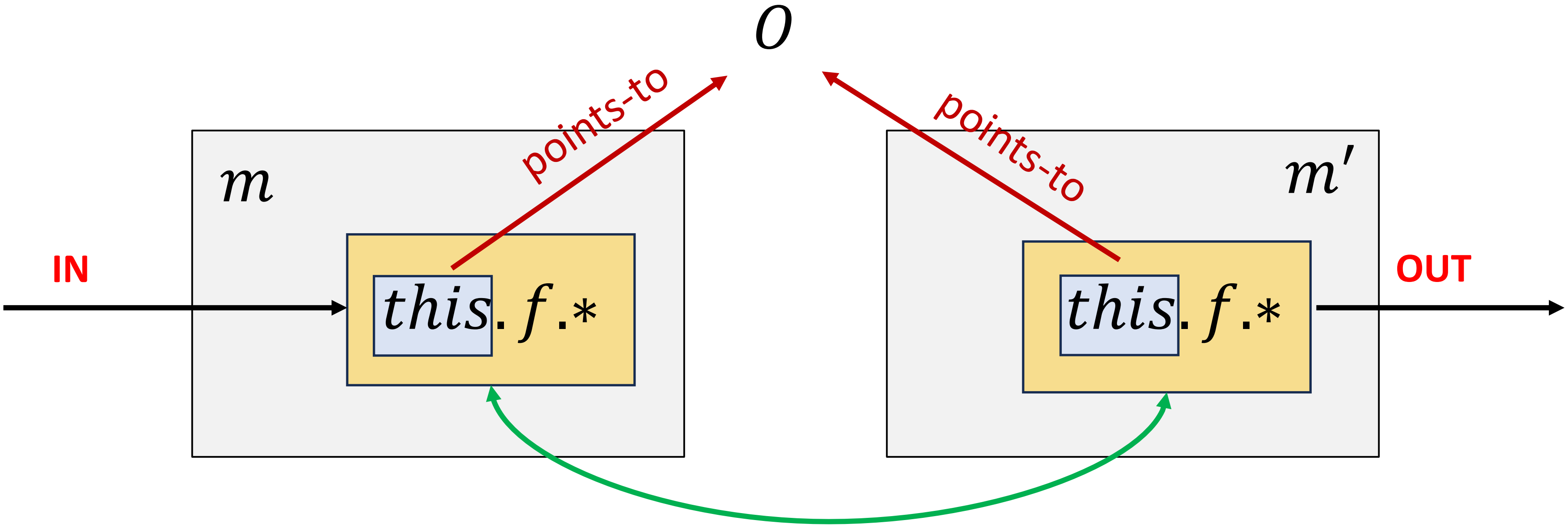
```
Input  : $P$ (Input Program)
Output : $\mathcal{I}$ (Context-Independent Objects)
begin
    Step 1: Find container objects in $P$ and collect them in containers.
    Step 2: Find context-dependent objects according to container-usage patterns.
        $\mathcal{D} \leftarrow \emptyset$ ;    // Initialize the set for collecting Context-dependent Objects
        foreach object $o \in$ containers do
            Step 2.1: if $o$ is an inner container then $\mathcal{D} \ni o$ ;
            Step 2.2: if $o$ is a factory-created container then $\mathcal{D} \ni o$ ;
            Step 2.3: if $o$ is a container wrapper then $\mathcal{D} \ni o$ ;
    Step 3: return the set of context-independent objects ($\mathcal{I} = \mathbb{H} \setminus \mathcal{D}$)
```

- **CI** objects in $\mathcal{I}$ will be used for **context debloating**

inhibit context combinations on CI elements

Container objects

- ❑An object O is a **container object** if it has at least
  - ○ One **pointer field** f,
  - ○ An **incoming value flow** s.t. $this^{m}.f.* = p\ (p \in \{p_i^{m}\})$, and
  - ○ An **outgoing value flow** s.t. $v.* = this^{m'}.f.*\ (v \in \{ret^{m'}, p_i^{m'}\})$
  - ○ m and m' are methods invoked on O



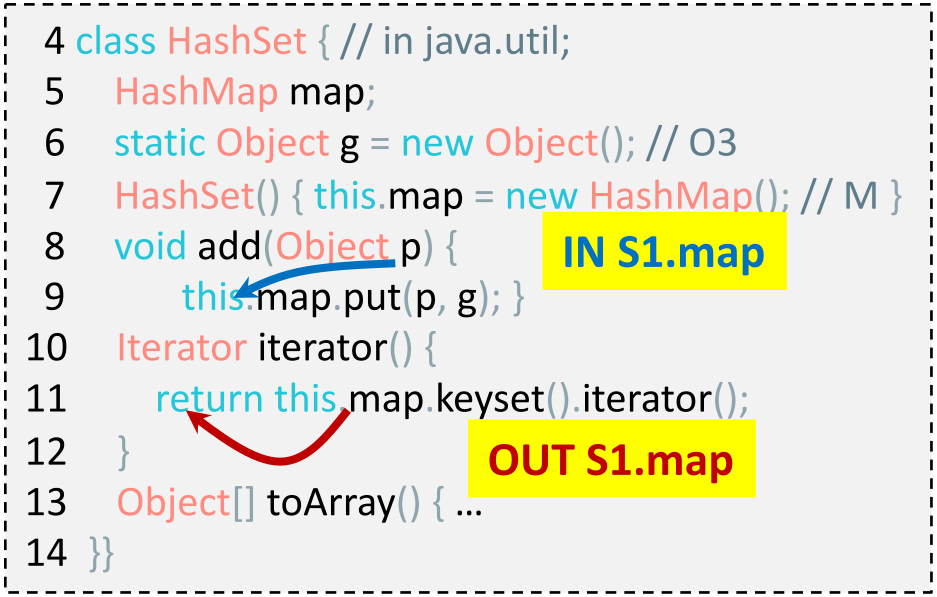
Aliases in the approximation of **1-limited** field access path

Container objects

- ❑ An object O is a **container object** if it has at least
  - ○ One **pointer field** f,
  - ○ An **incoming value flow** s.t. $this^{m}.f.* = p \; (p \in \{p_i^{m}\})$, and
  - ○ An **outgoing value flow** s.t. $v.* = \; this^{m'}.f.* \; (v \in \{ret^{m'}, p_i^{m'}\})$
  - ○ m and m' are methods invoked on O

- ❑ An example
  - ○ **S1** is a container object

```
HashSet s1 = new HashSet();// S1
s1.add(new Object()); // O1
Object o1 = s1.toArray()[0];
```



```
class HashSet { // in java.util;
  HashMap map;
  static Object g = new Object(); // O3
  HashSet() { this.map = new HashMap(); // M }
  void add(Object p) {
     this.map.put(p, g); }
  Iterator iterator() {
    return this.map.keyset().iterator();
  }
  Object[] toArray() { …
}}
```

IN S1.map

OUT S1.map

Container objects

❑Rules for identifying Container Objects

$$\frac{\begin{array}{c} O \in \mathbb{H} \quad t = \mathsf{typeof}(O) \quad t \text{ is an instance type} \\ f \in \mathtt{fields}(O) \quad t' = \mathsf{typeof}(f) \quad t' \in \mathsf{openTypes} \\ \mathtt{hasInFlow}\ (O, f) \quad \mathtt{hasOutFlow}\ (O, f) \end{array}}{O \in \mathtt{containers}} \qquad \frac{\begin{array}{c} O \in \mathbb{H} \quad t = \mathsf{typeof}(O) \\ t \text{ is an array type} \quad t \in \mathsf{openTypes} \end{array}}{O \in \mathtt{containers}}\ [\textsc{Con}]$$

$$\frac{\begin{array}{c} m \in \mathtt{methodsInvokedOn}(O) \quad f \in \mathtt{fields}(O) \\ p \in \mathtt{params}(m) \cap \mathtt{inParams}(f) \end{array}}{\mathtt{hasInFlow}(O, f)} \qquad \frac{\begin{array}{c} a.f = \_ \text{ is a store in method } m \quad O \in \overline{\mathtt{pts}}(a) \\ O \text{ not allocated in } m \quad a \notin assign^*(\mathtt{this}^m) \end{array}}{\mathtt{hasInFlow}(O, f)}\ [\textsc{In}]$$

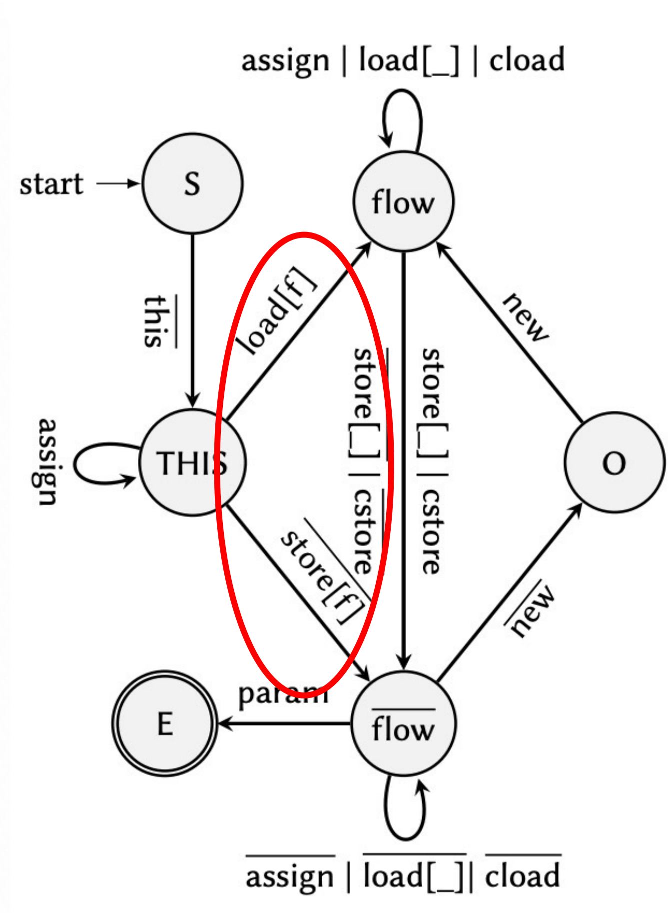
$$\frac{\begin{array}{c} m \in \mathtt{methodsInvokedOn}(O) \quad f \in \mathtt{fields}(O) \\ v \in \mathtt{paramsRet}(m) \cap \mathtt{outParamsRets}(f) \end{array}}{\mathtt{hasOutFlow}(O, f)} \qquad \frac{\begin{array}{c} \_ = a.f \text{ is a load in method } m \quad O \in \overline{\mathtt{pts}}(a) \\ O \text{ not allocated in } m \quad a \notin assign^*(\mathtt{this}^m) \end{array}}{\mathtt{hasOutFlow}(O, f)}\ [\textsc{Out}]$$

$$\frac{t \in \mathbb{T} \text{ is java.lang.Object}}{t \in \mathsf{openTypes}} \qquad \frac{t \in \mathbb{T} \text{ is an abstract type}}{t \in \mathsf{openTypes}} \qquad \frac{t \in \mathbb{T} \text{ is an interface type}}{t \in \mathsf{openTypes}}$$

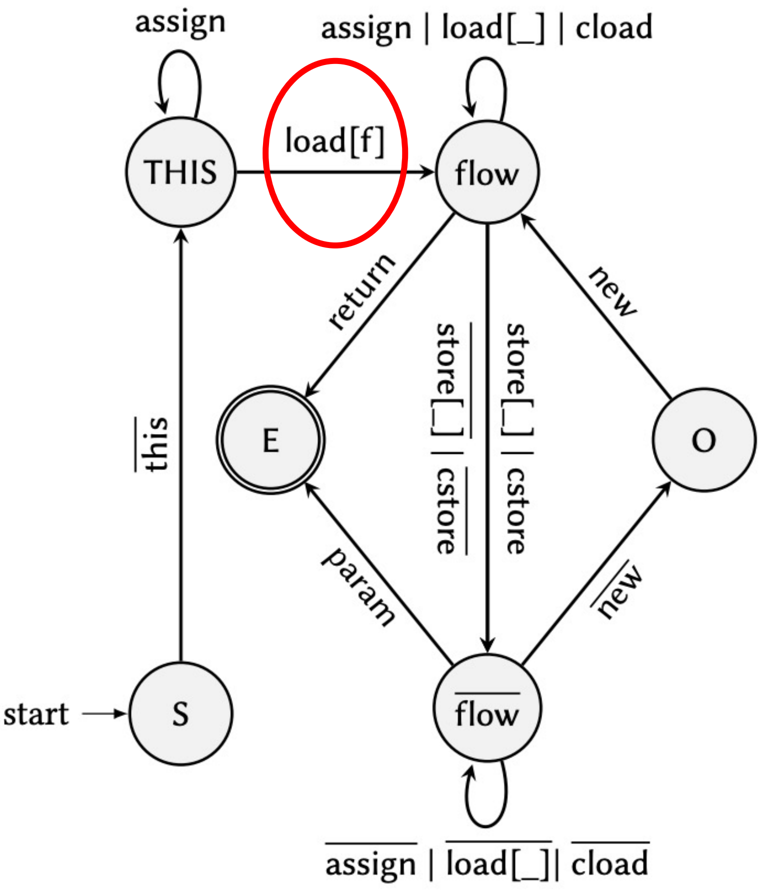
$$\frac{t \in \mathsf{openTypes}}{[t] \in \mathsf{openTypes}} \qquad \frac{t \in \mathbb{T} \quad f \in \mathtt{fields}(t) \quad \mathsf{typeof}(f) = t' \quad t' \in \mathsf{openTypes}}{t \in \mathsf{openTypes}}$$

Container objects

- ❑DFAs for computing incoming/outgoing value flows



$\mathcal{A}^{f}_{in}$

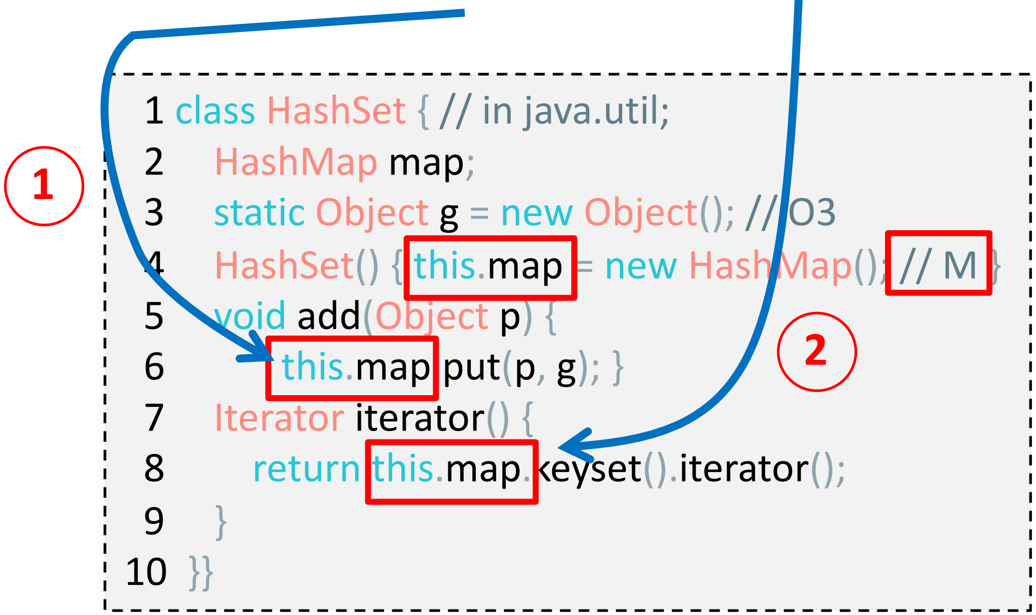


$\mathcal{A}^{f}_{out}$

- ❑1-limited field sensitive

Container-Usage Patterns

- ❑ Pattern I: Inner Containers
  - ○ Accessed by outer containers via a field
  - ○ Used by outer containers for storing and retrieving data



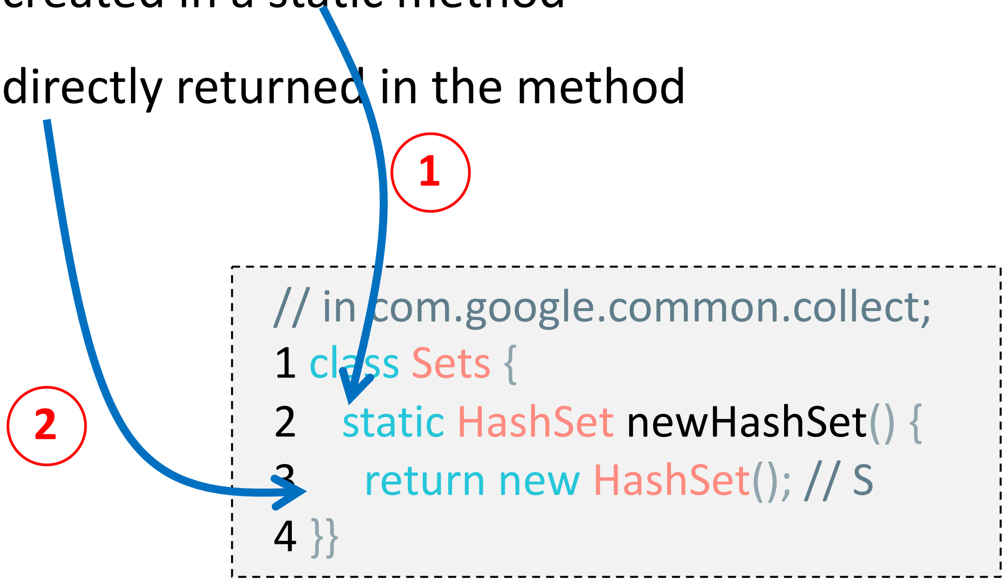
```
class HashSet { // in java.util;
    HashMap map;
    static Object g = new Object(); // O3
    HashSet() { this.map = new HashMap(); // M }
    void add(Object p) {
        this.map.put(p, g); }
    Iterator iterator() {
        return this.map.keyset().iterator();
    }
}}
```

- ❑ Inner containers are context-dependent
  - ○ Distinguish the data stored by its different outer container objects

Container-Usage Patterns

❑Pattern II: Factory-Created Containers

- created in a static method
- directly returned in the method



```
// in com.google.common.collect;
class Sets {
  static HashSet newHashSet() {
    return new HashSet(); // S
}}
```

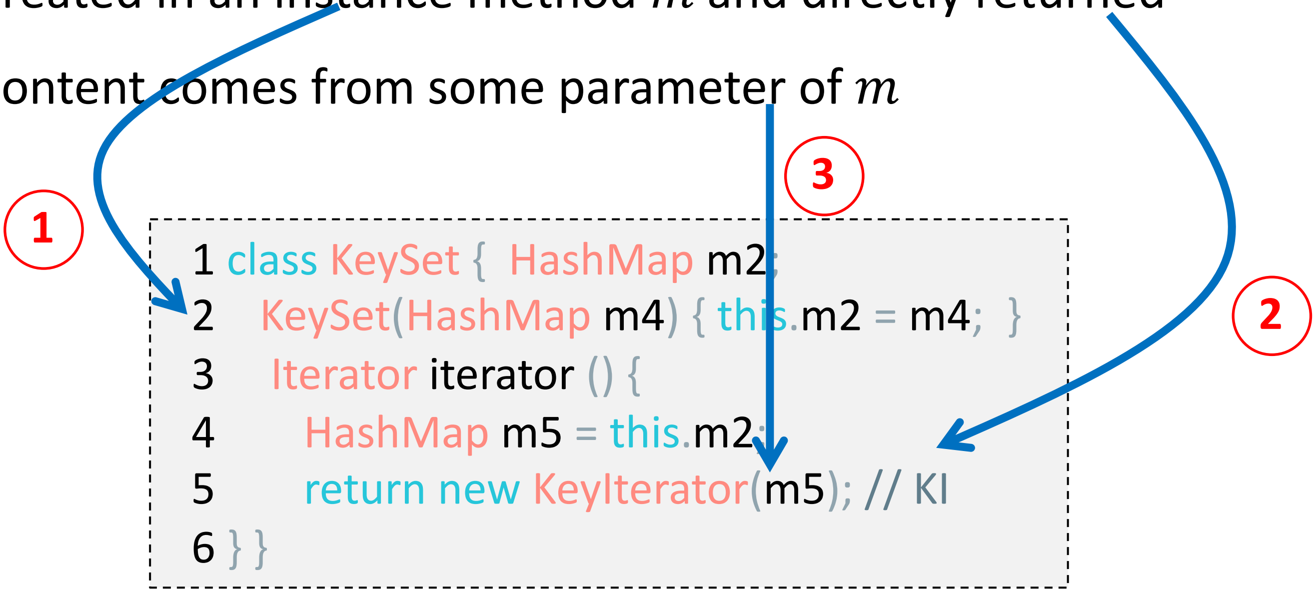
❑Factory-created containers are context-dependent

- Differentiate the data coming from different calling contexts

Container-Usage Patterns

❑Pattern III: Container Wrappers

- often are iterators, enumerators, …
- created in an instance method m and directly returned
- content comes from some parameter of m



```
class KeySet {  HashMap m2;
  KeySet(HashMap m4) { this.m2 = m4;  }
   Iterator iterator () {
      HashMap m5 = this.m2;
      return new KeyIterator(m5); // KI
} }
```

❑Container wrappers are context-dependent

- Differentiate the wrapped content coming from different calling contexts

Rules for Identifying Usage Patterns

- Rule for identifying inner containers

$$\frac{\begin{array}{c} O \in \mathbb{H} \quad m = \texttt{methodof}(O) \quad m \text{ is an instance method} \quad f \in \texttt{objectStoredInto}(O) \quad t = \texttt{typeof}(f) \\ t \in \texttt{openTypes} \quad O' \in \texttt{receiverObjects}(m) \quad \texttt{hasInFlow}(O', f) \quad \texttt{hasOutFlow}(O', f) \end{array}}{\texttt{isAnInnerContainer}(O)}$$

- Rule for identifying factory-created containers

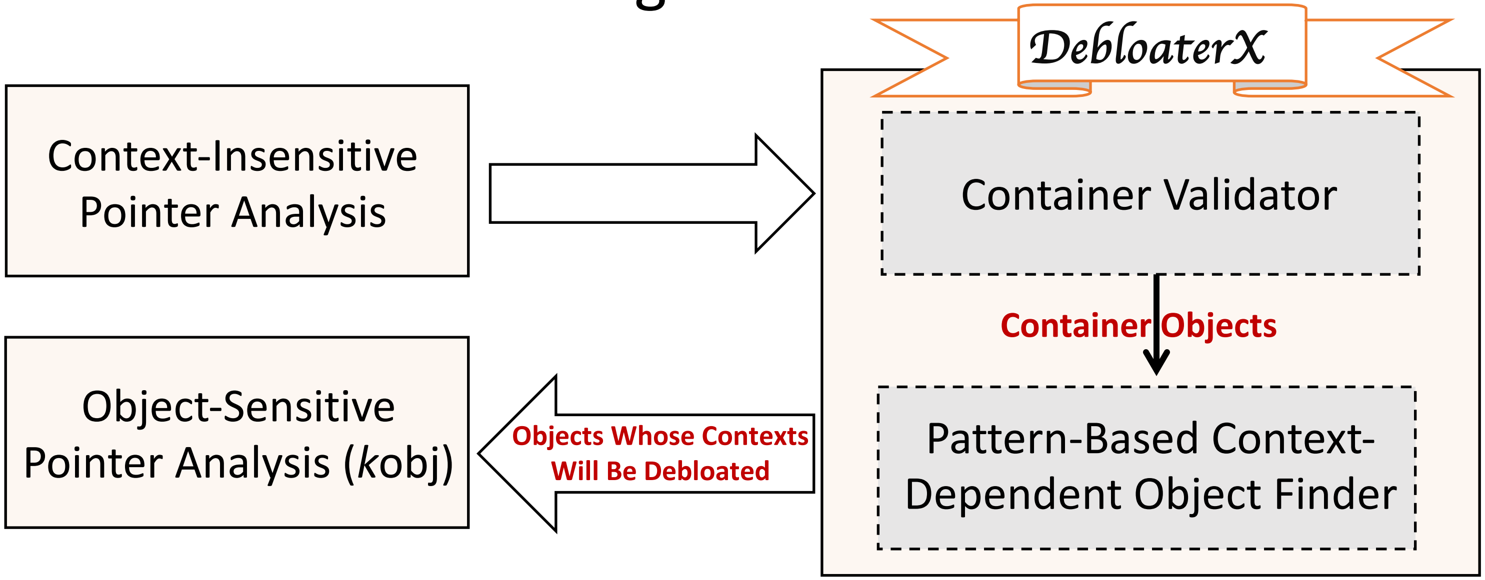
$$\frac{\begin{array}{c} m \text{ is a static method} \quad m = \texttt{methodof}(O) \\ \texttt{isDirectlyReturned}(\mathrm{O}) \end{array}}{\texttt{isAFactoryCreatedContainer}(O)} \qquad \frac{\begin{array}{c} O \xrightarrow{\text{new}} x \text{ is an edge in XPAG} \\ m = \texttt{methodof}(O) \quad \texttt{ret}^{m} \in \textit{assign}^{*}(x) \end{array}}{\texttt{isDirectlyReturned}(O)}$$

- Rule for identifying container wrappers

$$\frac{\begin{array}{c} m = \texttt{methodof}(O) \quad m \text{ is an instance method} \\ \texttt{isDirectlyReturned}(O) \quad \texttt{isContentFromParam}(O) \end{array}}{\texttt{isAContainerWrapper}(O)}$$

Workflow and Implementation

❑Workflow of DebloaterX-guided kOBJ



❑Implementation (~1500 LOC in Java)

- Source: https://github.com/DongjieHe/DebloaterX
- Artifact: https://hub.docker.com/r/hdjay2013/debloaterx
- Released in Qilin framework (https://qilinpta.github.io)

Evaluation

- Benchmarks
  - 5 dacapo 2006 benchmarks + JRE1.6
  - 2 real-world applications + JRE1.6
  - 5 dacapo-9.12 benchmarks + JRE1.8

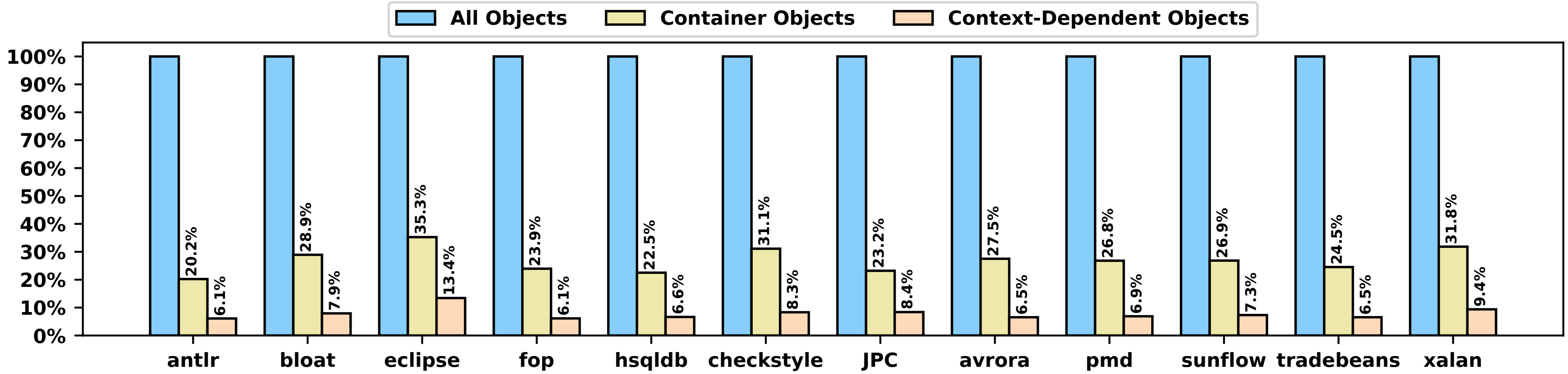
JPC

JREs

- Linux server with 512GB memory, 16 cores
- Time budget: 12 hours
- Metrics
  - Efficiency: *analysis time*
  - Precision: *#fail-casts, #reachable, #call-edges, #poly-calls*

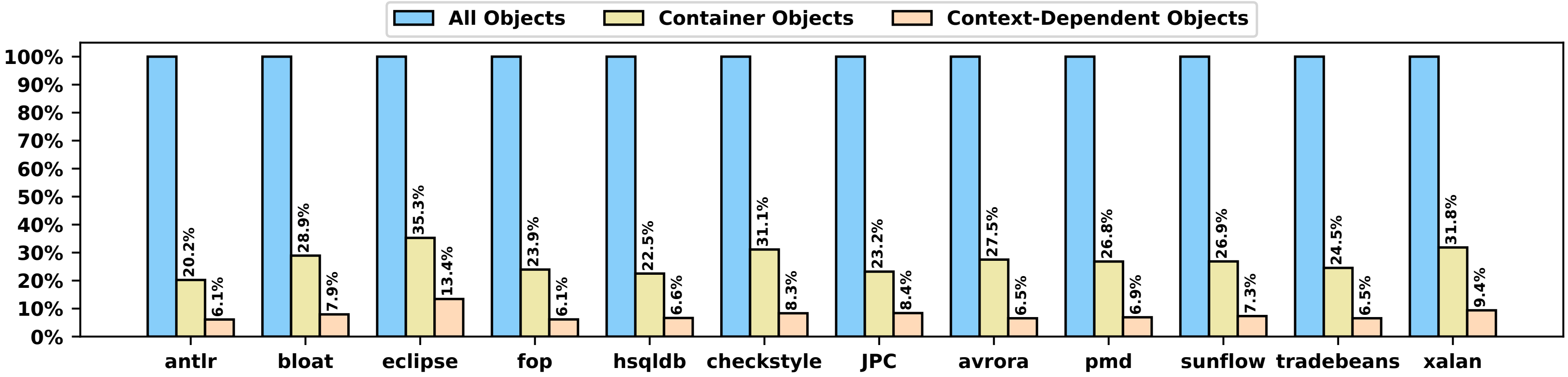
Evaluation: Objects Identification

- Containers: 26.6%, context-dependent objects: 7.6%

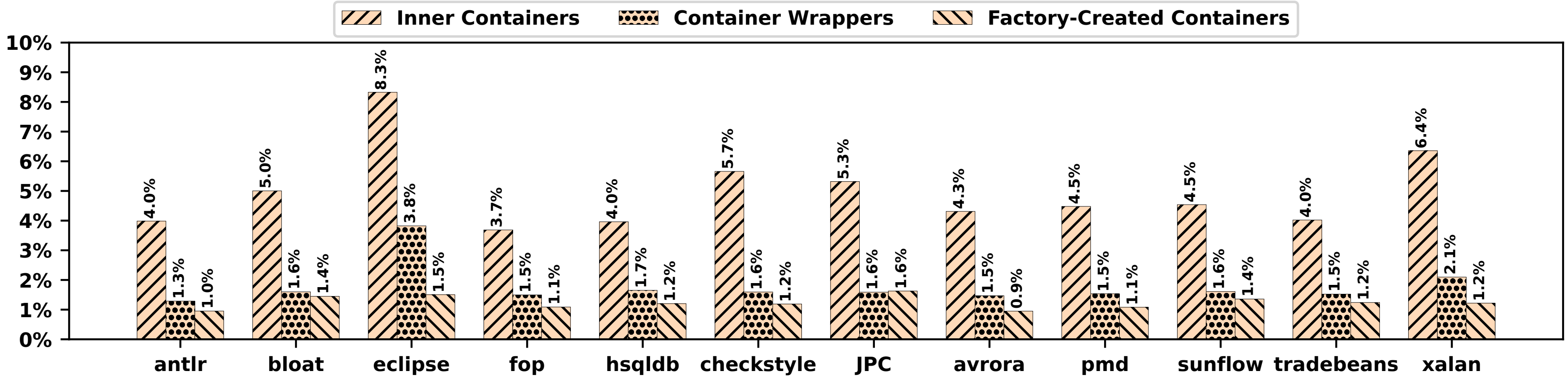


Evaluation: Objects Identification

- Containers: 26.6%, context-dependent objects: 7.6%

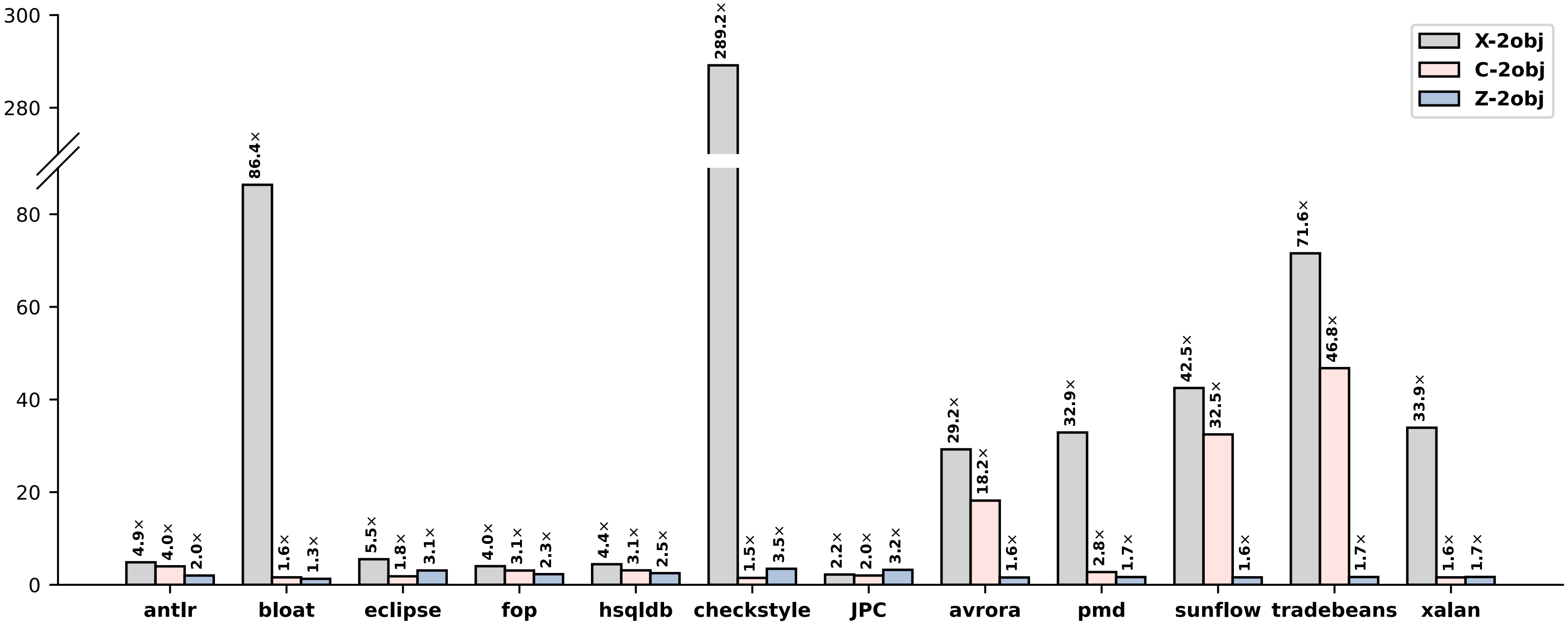


- Inner containers: 4.8% , factory-created containers: 1.2%, container wrappers: 1.7%



Evaluation: Efficiency

- 19.3x (150.2x) when k = 2 (3), faster than Conch and Zipper

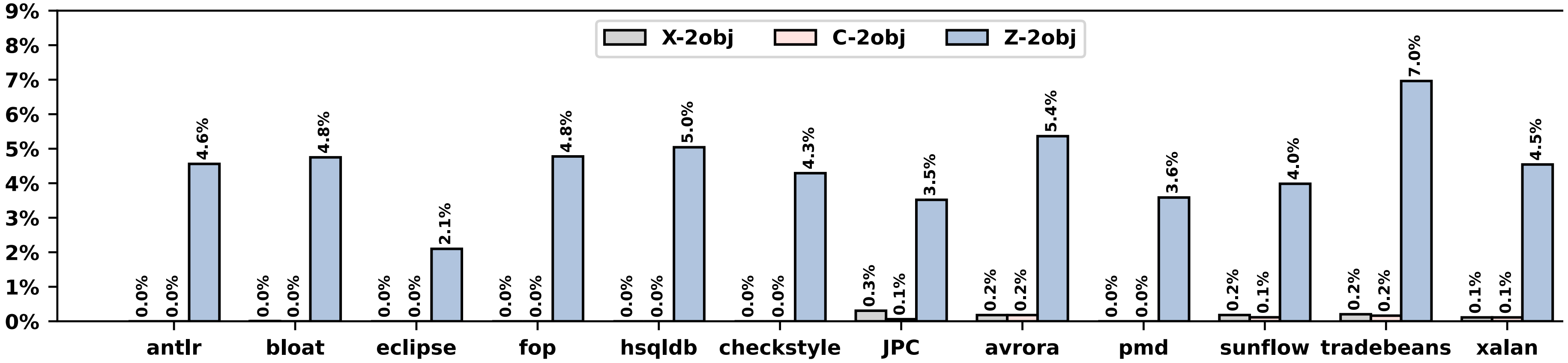


The enabled Speedup of **DebloaterX**, **Conch**, and **Zipper** over 2OBJ

- Better scalability
  - Scale 1 more than Conch, 6 more than Zipper
  - Scale 7 more than standard kOBJ

Evaluation: Precision

- ☐ Negligible precision loss (**<0.2%**), similar to Conch
- ☐ Precise than Zipper (**~4.5%** loss)



The precision loss of **DebloaterX**, **Conch**, and **Zipper**-guided 2OBJ over 2OBJ

Summary

- ❑ A new context debloating technique, DebloaterX
  - ○ Based on three container usage patterns
- ❑ enable kOBJ to be more efficient than state-of-the-art while preserving nearly all of the precision.

X-kOBJ: A precise yet efficient pointer analysis ☑

- ❑ Future work
  - ○ Investigate performance issues on extremely larger programs, e.g., eclipse
  - ○ Develop context-debloating techniques for other context-sensitivity

Q & A

Thank You!