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, and JINGLING XUE, University of New South Wales, Australia

In this paper, we introduce DEBLOATERX, a new approach for automatically identifying context-independent objects to debloat contexts in object-sensitive pointer analysis ($k$obj). Object sensitivity achieves high precision, but its context construction mechanism combines objects with their contexts indiscriminately. This leads to a combinatorial explosion of contexts in large programs, resulting in inefficiency. Previous research has proposed a context-debloating approach that inhibits a pre-selected set of context-independent objects from forming new contexts, improving the efficiency of $k$obj. However, this earlier context-debloating approach under-approximates the set of context-independent objects identified, limiting performance speedups.

We introduce a novel context-debloating pre-analysis approach that identifies objects as context-dependent only when they are potentially precision-critical to $k$obj based on three general container-usage patterns. Our research finds that objects containing no fields of "abstract" (i.e., open) types can be analyzed context-insensitively with negligible precision loss in real-world applications. We provide clear rules and efficient algorithms to recognize these patterns, selecting more context-independent objects for better debloating. We have implemented DEBLOATERX in the QILIN framework and will release it as an open-source tool. Our experimental results on 12 standard Java benchmarks and real-world programs show that DEBLOATERX selects 92.4% of objects to be context-independent on average, enabling $k$obj to run significantly faster (an average of 19.3× when $k = 2$ and 150.2× when $k = 3$) and scale up to 8 more programs when $k = 3$, with only a negligible loss of precision (less than 0.2%). Compared to state-of-the-art alternative pre-analyses in accelerating $k$obj, DEBLOATERX outperforms ZIPPER significantly in both precision and efficiency, and outperforms CONCH (the earlier context-debloating approach) in efficiency substantially while achieving nearly the same precision.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**.

Additional Key Words and Phrases: pointer analysis, context debloating, container usage patterns

ACM Reference Format:
Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. 2023. A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 256 (October 2023), 30 pages. https://doi.org/10.1145/3622832

1 INTRODUCTION

Pointer analysis, which statically and conservatively estimates which heap objects pointer variables may refer to at runtime, is a key component of various software analysis techniques such as compiler optimization [Phulia et al. 2020], bug detection [Cai et al. 2022], program understanding [Sridharan et al. 2007], program verification [Garcia-Contreras et al. 2022], and symbolic execution [Trabish et al. 2020]. The success of these techniques ultimately depends on the precision and efficiency of the underlying pointer analysis algorithm used.

Authors' address: Dongjie He, dongjieh@cse.unsw.edu.au; Yujiang Gui, yujiang.gui@unsw.edu.au; Wei Li, liwei@cse.unsw.edu.au; Yonggang Tao, yonggangtao@cse.unsw.edu.au; Changwei Zou, changwei.zou@unsw.edu.au; Yulei Sui, y.sui@unsw.edu.au; Jingling Xue, jingling@cse.unsw.edu.au, University of New South Wales, Sydney, NSW, Australia, 2052.

2475-1421/2023/10-ART256
https://doi.org/10.1145/3622832

For object-oriented languages like Java, object-sensitivity [Milanova et al. 2002, 2005] is an essential technique for ensuring high precision in Java pointer analyses and is widely adopted in several Java pointer analysis frameworks, such as Qilin [He et al. 2022], Doop [Bravenboer and Smaragdakis 2009], and Wala [WALA 2023]. Under k-limiting, the *method context* of a k-object-sensitive analysis (with a $(k-1)$-context-sensitive heap), denoted k*obj*, is represented by a sequence of k context elements, $[o_1, ..., o_k]$, where o_1 is the receiver object of some method being analyzing and o_i is a receiver object of the method where o_{i-1} is allocated [Smaragdakis et al. 2011].

However, scaling $k$obj for reasonably large programs when $k \geqslant 3$ can be difficult, and even when it is scalable, it can be time-consuming [He et al. 2021b; Jeon et al. 2018; Li et al. 2018; Thiessen and Lhoták 2017]. To address this issue, He et al. [2021a, 2023a] recently proposed a context-debloating approach, which functions as a pre-analysis, to debloat contexts for $k$obj. They identified the primary reason for the inefficiency of $k$obj as the context combinatorial explosion issue caused by its context construction mechanism, which combines objects indiscriminately with their contexts to form new contexts. However, in practice, the majority of objects in a program are context-independent rather than context-dependent. Allowing these objects to combine with contexts to form new contexts only serves to increase the number of contexts that need to be analyzed, without providing any significant precision benefits. The definitions and explanations of context-dependent and context-independent objects can be found in Section 2. Unlike selective context-sensitivity approaches (which are also performed as a pre-analysis) [Hassanshahi et al. 2017; He et al. 2021b; Jeong et al. 2017; Li et al. 2018, 2020; Lu et al. 2021a; Lu and Xue 2019; Smaragdakis et al. 2014], which aim to reduce the number of methods or program elements that require context-sensitivity, context-debloating approaches are theoretically more advantageous in boosting the performance of $k$obj since they can significantly reduce the number of contexts formed for not only context-independent objects but also context-dependent objects.

To improve context-debloating even further, the challenge is to design a pre-analysis that can identify as many context-independent objects (or as few context-dependent objects) as possible while also being both precise and efficient. However, verifying the context-dependability of an object by checking two CFL-reachability-based necessary conditions (i.e., CFL-P1 and CFL-P2 introduced in [Lu et al. 2021a]) is theoretically undecidable [Reps 2000]. Therefore, determining which objects are context-dependent requires approximations. The first context-debloating technique introduced in [He et al. 2021a, 2023a], Conch, approximates the two necessary conditions for context-dependability using three linear verifiable conditions. While Conch is already efficient, its field-insensitive algorithm selects context-dependent objects highly over-approximately, which limits its potential to further improve the performance of $k$obj.

We present DebloaterX, a novel context-debloating approach for $k$obj based on container-usage patterns. Our approach is designed to leverage the existence of real-world usage patterns, recognizing that despite the theoretical complexity of determining context-dependability, the usage patterns of context-dependent objects are finite. We identify three general container-usage patterns that dominate the real-world distribution of context-dependent objects: (1) *inner containers*, which save data from their outer containers, (2) *factory-created containers*, which are created by a factory method and used under different contexts to store and retrieve data, and (3) *container wrappers*, such as iterators and enumerators, that encapsulate a container and provide APIs for retrieving data. To efficiently and accurately identify these patterns, we have developed a set of efficient rules and algorithms that support a certain degree of field sensitivity while controlling pre-analysis time within acceptable limits. Our approach utilizes a specially designed graph structure called XPAG, which over-approximates value flows at virtual calls. We apply a set of finite-state automata to independently check the value flows of all object fields, allowing for parallelism and supporting 1-limited field sensitivity. By over-approximating value flows during the design of XPAG, we ensure

```
 1 class Client { // handcrafted
 2   void foo() {
 3     Object o1 = new Object(); // O1
 4     HashSet s1 = Sets.newHashSet();
 5     s1.add(o1);
 6     Object[] a1 = s1.toArray();
 7     Object v1 = a1[0];
 8   }
 9   void bar() {
10     Object o2 = new Object(); // O2
11     HashSet s2 = Sets.newHashSet();
12     s2.add(o2);
13     Object[] a2 = s2.toArray();
14     Object v2 = a2[0];
15 }}
16 // in com.google.common.collect;
17 class Sets {
18   static HashSet newHashSet() {
19     return new HashSet(); // S
20 }}
21 class HashSet { // in java.util;
22   HashMap map;
23   static Object g = new Object(); // O3
24   HashSet() {
25     this.map = new HashMap(); // M
26   }
27   void add(Object p) {
28     this.map.put(p, g);
29   }
30   Iterator iterator() {
31     return this.map.keySet().iterator();
32   }
33   Object[] toArray() {
34     Object[] r = new Object[5]; // [O]
35     r[0] = this.iterator().next();
36     return r;
37 }}
38 class HashMap { // in java.util
39   Entry[] table;
40   Set keySet;
41   HashMap() {
42     this.table = new Entry[5]; // [E]
43   }
44   Set keySet() {
45     Set s = this.keySet;
46     if (s == null) {
47       s = new KeySet(this); // KS
48       this.keySet = s;
49     }
50     return s;
51   }
52   void put(Object q1, Object q2) {
53     this.table[0] = new Entry(q1, q2); // E
54   }
55   class KeyIterator {
56     HashMap m1;
57     KeyIterator(HashMap m3) { this.m1 = m3; }
58     Object next() {
59       return this.m1.table[0].key;
60   }}
61   class Entry {
62     Object key, value;
63     Entry(Object p1, Object p2) {
64       this.key = p1; this.value = p2
65   }}
66   class KeySet {
67     HashMap m2;
68     KeySet(HashMap m4) { this.m2 = m4; }
69     Iterator iterator() {
70       HashMap m5 = this.m2;
71       return new KeyIterator(m5); // KI
72 }}}
73 new Client().foo(); // C1
74 new Client().bar(); // C2
```

Fig. 1. An example, where all objects except O1, O2, O3, C1, and C2 are context-dependent. S is a factory-created container, M, [E] and E are inner containers, and [O] and KI are container wrappers. KS is both an inner container and a container wrapper. Lines 73-74 in the orange box may be analyzed under contexts $c_1, ..., c_n$.

that automaton checking can be performed in time linear in the program statement size. To the best of our knowledge, our approach is the first pre-analysis to support field sensitivity within reasonable analysis time for accelerating $k$obj while causing it to suffer from minor precision loss.

We have implemented DebloaterX in Qilin, a Java pointer analysis framework designed for supporting fine-grained context-sensitivity [He et al. 2022], and evaluated it on 12 standard Java benchmarks and real-world programs. Our results demonstrate that DebloaterX identifies an average of 92.4% context-independent objects, enabling $k$obj to run significantly faster with speedups ranging from one to two orders of magnitude. Specifically, DebloaterX speeds up 2obj by 19.3× on average and 3obj by 150.2× on average. In addition, DebloaterX enables 3obj to scale to 8 more programs. The precision loss caused by DebloaterX is almost negligible, with less than 0.2% reduction in precision. We have also compared DebloaterX with two state-of-the-art pre-analyses for accelerating $k$obj: Zipper [Li et al. 2018, 2020] and Conch [He et al. 2021a, 2023a]. Our results indicate that DebloaterX outperforms Zipper significantly in both precision and efficiency and outperforms Conch in efficiency significantly while achieving nearly the same precision.

In summary, this paper makes the following main contributions:

- The identification of three general container-usage patterns that can potentially identify almost all context-dependent objects for $k$obj in real-world Java programs (Section 2).
- The introduction of DEBLOATERX, a precise yet efficient context-debloating approach for $k$obj, which leverages these three container-usage patterns (Section 4).
- The implementation of DEBLOATERX in QILIN and its open-source release at https://github.com/DongjieHe/DebloaterX.git.
- The extensive evaluation of DEBLOATERX, which demonstrates its effectiveness and practical significance for real-world programs, outperforming state-of-the-art alternatives (Section 5).

2 DEBLOATERX: AN OVERVIEW

We use an example Java program, as shown in Figure 1, which is abstracted from real-world code. This program serves to illustrate the challenges of verifying an object's context-dependability or context-independability. We also aim to demonstrate the limitations of existing work while reviewing object sensitivity and context debloating for $k$obj. The example may appear slightly complex, but it is designed to be complex enough to motivate our DEBLOATERX approach.

In this work, $k$obj operates on an intermediate representation of a Java program where generic types have been erased and are therefore not directly considered by $k$obj. As an example, in Figure 1, `HashSet<T>` is simplified to `HashSet<Object>` or simply `HashSet` for ease of understanding.

Example Program. In Figure 1, there are seven classes in total. The `Sets` class (lines 16-20) is from the third-party library Google Collect, while the `HashSet` class (lines 21-37) and `HashMap` class (lines 38-72, including its three inner classes: `KeyIterator` (lines 55-60), `Entry` (lines 61-65), and `KeySet` (lines 66-72)) are from the standard JDK library. The handcrafted class `Client` (lines 1-15) includes two logically equivalent methods, `foo()` and `bar()`. In `foo()` (`bar()`), the object `O1` (`O2`) created in line 3 (10) is added into a `HashSet` (i.e., `S`, created by a call to the factory method `newHashSet()` in line 4 (line 11)) and later loaded into `v1` (`v2`) after a call to the `toArray()` method in line 6 (13) and followed by a load. In the orange box (lines 73-74), two `Client` objects, `C1` and `C2`, are created and used as the receiver object to invoke `foo()` and `bar()`, respectively.

Object Sensitivity. Figure 2 depicts the Object Allocation Graph (OAG) [Tan et al. 2016] for our example. Here, an edge $n_1 \longrightarrow n_2$ indicates that object n_1 is a receiver object of a method where n_2 is allocated. Assuming for now that the code in the orange box resides in `main()` and thus analyzed only once under the empty context [ ], we can ignore all the OAG edges originating from $c_1, ..., c_n$. In $k$obj, a method can be analyzed under different calling contexts determined by the paths to its receiver objects. For example, `add()` (lines 27-29) is invoked in lines 5 and 12, both with `S` as its receiver object. As there are two paths to object `S`, `add()` will be analyzed separately under two different contexts, $[\mathtt{S}, \mathtt{C1}]$ and $[\mathtt{S}, \mathtt{C2}]$. For this particular example, all other methods except `foo()` and `bar()` will also be analyzed under two different contexts. Let $\mathtt{pts}(n, c)$ be the set of context-sensitive objects pointed by a pointer variable n under context c. Then, we have $\mathtt{pts}(\mathtt{v1}, [\mathtt{C1}]) = \mathtt{pts}([\mathtt{O}].\mathtt{arr}, [\mathtt{S}, \mathtt{C1}]) = \mathtt{pts}(\mathtt{E.key}, [\mathtt{M}, \mathtt{S}, \mathtt{C1}]) = \mathtt{pts}(\mathtt{q1}, [\mathtt{M}, \mathtt{S}, \mathtt{C1}]) = \mathtt{pts}(\mathtt{p}, [\mathtt{S}, \mathtt{C1}]) = \mathtt{pts}(\mathtt{o1}, [\mathtt{C1}]) = \{\langle \mathtt{O1}, [\mathtt{C1}]\rangle\}$, and $\mathtt{pts}(\mathtt{v2}, [\mathtt{C2}]) = \mathtt{pts}([\mathtt{O}].\mathtt{arr}, [\mathtt{S}, \mathtt{C2}]) = \mathtt{pts}(\mathtt{E.key}, [\mathtt{M}, \mathtt{S}, \mathtt{C2}]) = \mathtt{pts}(\mathtt{q1}, [\mathtt{M}, \mathtt{S}, \mathtt{C2}]) = \mathtt{pts}(\mathtt{p}, [\mathtt{S}, \mathtt{C2}]) = \mathtt{pts}(\mathtt{o2}, [\mathtt{C2}]) = \{\langle \mathtt{O2}, [\mathtt{C2}]\rangle\}$, where `arr` is a special field introduced for modeling an array object monolithically as is standard [Sridharan and Bodík 2006; Sridharan et al. 2005]. Therefore, `v1` only points to `O1` and `v2` only points to `O2` as expected. However, if any object in $\{\mathtt{S}, \mathtt{M}, [\mathtt{O}], [\mathtt{E}], \mathtt{E}, \mathtt{KS}, \mathtt{KI}\}$ is analyzed context-insensitively, then `v1` (`v2`) will be found to also point to `O2` (`O1`) spuriously.

Context-Dependability and Context-Independability. Let us introduce the formal definitions for these two concepts. Consider applying $k$obj to analyze a program P that includes an object o. We define $k\text{obj}^o$ as a version of $k$obj that analyzes object o context-insensitively while analyzing

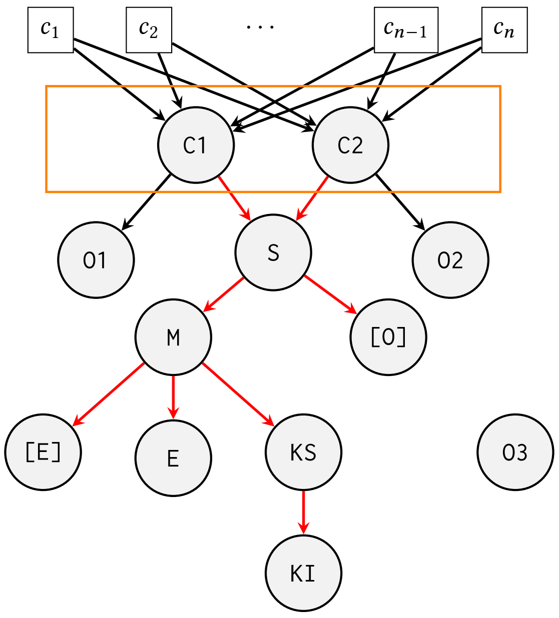


Fig. 2. The object allocation graph for the program shown in Figure 1 (with $c_1, \ldots, c_n$ being the contexts of C1 and C2), where only the red edges will remain after context debloating has been performed.

all other objects in P context-sensitively. For clarity, we write $pts_{k\text{obj}}(v, c)$ to represent the context-sensitive points-to set computed by $k$obj for a pointer variable v under a context c (i.e., $pts(v, c)$ defined earlier for $k$obj in this section). Let $pts_{k\text{obj}}(v)$ be the corresponding points-to set with the context information being dropped, given by $pts_{k\text{obj}}(v) = \cup_{c \in ctx(v)} \{o \mid \langle o, c' \rangle \in pts_{k\text{obj}}(v, c)\}$, where $ctx(v)$ is the set of all contexts of v. Similarly, we define $pts_{k\text{obj}^o}(v, c)$ and $pts_{k\text{obj}^o}(v)$.

Now, object o is said to be *context-independent* if and only if for every pointer variable v in P, $pts_{k\text{obj}}(v) = pts_{k\text{obj}^o}(v)$ holds for all values of k. Conversely, object o is *context-dependent* if and only if there exists a pointer variable v in P such that $pts_{k\text{obj}}(v) \neq pts_{k\text{obj}^o}(v)$ for some value of k.

Context Debloating. The idea behind context debloating is simple: objects in a program are categorized as either context-dependent or context-independent. During $k$obj, context-independent objects are analyzed context-insensitively and are not differentiated under different calling contexts. This approach effectively reduces the number of contexts that would otherwise be formed and used in analyzing a method, significantly improving the efficiency and scalability of $k$obj.

Let us return to Figure 1 assuming that the code in the orange box is now analyzed under n different contexts, $c_1, \ldots, c_n$. If context debloating is not used, objects C1 and C2 will combine with every context c_i to form new contexts $[\texttt{C1}, c_i]$ and $[\texttt{C2}, c_i]$ used for analyzing `foo()` and `bar()`, respectively, causing each of these two methods to be analyzed a total of n times, and every other method in the program to be analyzed a total of $2n$ times. However, such over analysis brings no benefits for precision. Suppose we know that O1, O2, O3, C1, and C2 in the program are context-independent, while the other objects are context-dependent. By using context debloating, all edges to a context-independent object in the OAG given in Figure 2 are removed, resulting in a debloated OAG consisting of only the red edges. This leads to a more efficient $k$obj while achieving the same precision when analyzing our example with its contexts debloated this way.

Challenges. Developing a precise context-debloating technique is a theoretically challenging task. Lu et al. [2021a] have recently proposed two necessary conditions (as shown in their Figure 2) for determining whether an object should be analyzed context-sensitively or not. However, verifying these two necessary conditions requires computing the reachability problem of the intersection of two interleaved context-free languages (CFLs), which is an undecidable problem [Reps 2000].

Limitations of Existing Work. Eagle [Lu et al. 2021a; Lu and Xue 2019] weakens the two necessary conditions by regularizing one of the two CFLs involved, leading to a conservative identification of almost all non-trivial context-independent objects as context-dependent objects. While Eagle maintains precision, it can only provide a speedup of approximately 1.5× for $k$obj [He et al. 2022]. In contrast, Conch [He et al. 2021a, 2023a], which is the first context-debloating approach for $k$obj, uses three linearly verifiable conditions to determine an object's context-dependability. While Conch is already highly efficient, it selects context-dependent objects in an over-approximate manner, which limits its ability to further improve the performance of $k$obj.

Consider object `C1` in our example program, which both Eagle and Conch can correctly identify as context-independent. However, if we make the following slight modification, both approaches will fail to do so, despite no change to the context-dependability of any object. Suppose we introduce a new field "`Object f`" in `Client`, add a parameter "`Object t1`" and two statements "`this.f = t1`" and "`t2 = this.f`" to `foo()`, and encapsulate line 73 in a new method that returns `C1`. In this modified example, Conch will falsely identify `C1` as context-dependent, as it satisfies all three of its verification conditions, but the value from `t1` can never flow out of `foo()` (thus violating the set of two necessary conditions proposed in Lu et al. [2021a] as mentioned above).

Our solution. Due to the theoretical complexity of verifying an object's context-dependability, we propose to rely on real-world usage patterns to identify context-dependent objects, which has not been done before. Our solution is a novel approach called DebloaterX, which leverages three general container-usage patterns to debloat contexts for $k$obj.

In the example shown in Figure 1, there are seven context-dependent objects: `S`, `M`, `[O]`, `[E]`, `KS`, `E`, and `KI`. Each of these objects is a container, as defined in Definition 1, which stores data that is first retrieved from outside the container and later accessed outside the container. For instance, `M` is a container because it stores `O1` (`O2`) in `M.table.arr.key` and later provides it to `v1` (`v2`). Similarly, `KI` is also a container because it stores `M` (along with all the data in `M`) in `KI.m1` and later exposes `KI.m1.key` to `r[0]` in line 35. As for array objects, `[O]` and `[E]`, they are naturally containers because they are frequently subject to stores and loads. Our definition of containers differs from informal definitions used in the literature, as we require the existence of specific incoming and outgoing value flows outside the containers via their field accesses (Definition 1).

Furthermore, we have discovered that container objects are context-dependent only when used in terms of certain usage patterns. Based on our observations, we have identified three general container usage patterns that are associated with context-dependent objects:

- ***Inner Containers.*** These are containers that are typically accessed by their outer containers via a field (or precisely a field access path) and are used by their outer containers for storing and retrieving data. Example inner containers in Figure 1 include `M`, `[E]`, `KS`, and `E`.
- ***Factory-Created Containers.*** Some containers, such as `S` in Figure 1, are created by factory methods. These methods are typically static and invoked under different calling contexts. By ensuring context-dependability of these containers, we can guarantee that the data stored in them under different contexts will not be conflated, preserving the precision of the analysis.
- ***Container Wrappers.*** These are objects that wrap around other containers and are returned directly by their allocating methods, responsible for storing data in their fields and providing APIs for accessing the data. Examples in Figure 1 include `[O]`, `KS`, and `KI`. In general, iterator

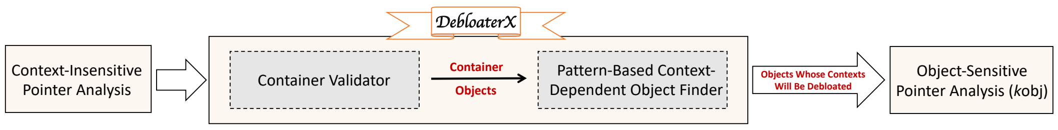

DebloaterX
Context-Insensitive Pointer Analysis
Container Validator
Container Objects
Pattern-Based Context-Dependent Object Finder
Objects Whose Contexts Will Be Debloated
Object-Sensitive Pointer Analysis (kobj)

Fig. 3. Overview of DebloaterX for debloating contexts for *k*obj.

and enumerator objects, which are prevalent in the JDK library, third-party libraries, and regular Java applications, are representative examples of container wrappers.

Note that there may be some overlap between inner containers and container wrappers, as shown by the existence of KS in Figure 1 being classified as both. However, as our evaluation in Section 5 demonstrates, such overlap is not necessarily common, in practice.

Although we anticipated that the number of container-usage patterns would be finite, we were surprised by the small number of patterns we have discovered. Despite this, identifying only these three patterns is sufficient to preserve the 99.8% precision of *k*obj, in practice (Section 5).

Figure 3 provides a high-level overview of DebloaterX. DebloaterX first identifies container objects in a program and then checks whether their usage matches any of the three patterns outlined earlier to determine their context-dependability. DebloaterX utilizes points-to information that is pre-computed by a context-insensitive pointer analysis and serves to improve the efficiency of *k*obj by instructing it to analyze the context-independent objects in a context-insensitive manner.

For our example given in Figure 1 and the modified version discussed above, DebloaterX can accurately distinguish between context-independent and context-dependent objects.

The precise yet efficient identification of container objects and the three usage patterns remains a significant challenge. To address this, we have developed several sophisticated finite-state automata, along with efficient rules and algorithms. We defer their technical details to Section 4.

3 PRELIMINARIES

We introduce some notations used in the paper (Section 3.1) and provide a brief review of a formalization of object-sensitive pointer analysis (*k*obj) with support for context debloating (Section 3.2). This forms the basis for understanding our pre-analysis, DebloaterX, introduced in Section 4.

3.1 Notations

Given a Java program P, let $\mathbb{T}, \mathbb{H}, \mathbb{M}, \mathbb{F}, \mathbb{V}, \mathbb{L}$ be the domains of types, heap objects, methods, instance fields, variables, and statement labels, respectively. For *k*obj, we write $\mathbb{C} = \mathbb{H}^0 \cup \mathbb{H}^1 \cup \mathbb{H}^2 \cdots$ to represent the domain of contexts. Given a context $c = [e_1, \cdots, e_n] \in \mathbb{C}$ and a context element $e \in \mathbb{H}$, we write $e :: c$ for $[e, e_1, \cdots, e_n]$ and $\lceil c \rceil_k$ for $[e_1, \cdots, e_k]$. Given an instance method m, we write $\texttt{this}^m$, p_i^m and $\texttt{ret}^m$ for the "this" variable, i-th parameter and return variable (i.e., a unique variable introduced for storing all returned values) of m, respectively. Static fields and methods are omitted. Static fields can be treated as regular variables, even though they are always analyzed context-insensitively. Static methods are handled in the standard manner [He et al. 2022].

In addition, the following seven auxiliary functions are used: (1) fields : $\mathbb{T} \cup \mathbb{H} \hookrightarrow \mathcal{P}(\mathbb{F})$ returns a set of fields accessed by a heap object or any instance object of a class, (2) methodof : $\mathbb{H} \hookrightarrow \mathbb{M}$ gives the allocating method of a heap object, (3) typeof : $\mathbb{F} \cup \mathbb{H} \hookrightarrow \mathbb{T}$ gives the declaring type of an instance field or the dynamic type of a heap object, (4) $\overline{\texttt{pts}}$: $\mathbb{V} \cup \mathbb{H} \times \mathbb{F} \hookrightarrow \mathcal{P}(\mathbb{H})$ records the points-to information found context-insensitively (e.g., by a context-insensitive pointer analysis like Spark [Lhoták and Hendren 2003]) for a variable or an object's field, (5) methodsInvokedOn : $\mathbb{H} \hookrightarrow \mathcal{P}(\mathbb{M})$ returns the set of methods that can be invoked on a receiver object (computed by, say, Spark), (6) params : $\mathbb{M} \hookrightarrow \mathcal{P}(\mathbb{V})$ returns the set of parameters (including the *this* variable) of a method, and

$$\frac{\begin{array}{c} x = \text{new T} \; // \; O \text{ in } m \quad c \in \texttt{contexts}(m) \\ \boxed{c' = \text{if } O \in \mathcal{I} \text{ then } [\,] \text{ else } \lceil c \rceil_{k-1}} \end{array}}{\langle O, c' \rangle \in \texttt{pts}(x, c)} \; [\textsc{New}] \qquad \frac{\begin{array}{c} x = y \text{ in } m \\ c \in \texttt{contexts}(m) \end{array}}{\texttt{pts}(y, c) \subseteq \texttt{pts}(x, c)} \; [\textsc{Assign}]$$

$$\frac{\begin{array}{c} x = y.f \text{ in } m \quad c \in \texttt{contexts}(m) \\ \langle O, c' \rangle \in \texttt{pts}(y, c) \end{array}}{\texttt{pts}(O.f, c') \subseteq \texttt{pts}(x, c)} \; [\textsc{Load}] \qquad \frac{\begin{array}{c} x.f = y \text{ in } m \quad c \in \texttt{contexts}(m) \\ \langle O, c' \rangle \in \texttt{pts}(x, c) \end{array}}{\texttt{pts}(y, c) \subseteq \texttt{pts}(O.f, c')} \; [\textsc{Store}]$$

$$\frac{\begin{array}{c} l : x = y.m'(a_1, ..., a_n) \text{ in } m \quad c \in \texttt{contexts}(m) \\ \langle O, c' \rangle \in \texttt{pts}(y, c) \quad m'' = \texttt{dispatch}(O, l) \quad c'' = O :: c' \end{array}}{\begin{array}{c} c'' \in \texttt{contexts}(m'') \quad \forall i \in [1, n] : \texttt{pts}(a_i, c) \subseteq \texttt{pts}(p_i^{m'}, c'') \\ \langle O, c' \rangle \in \texttt{pts}(\texttt{this}^{m''}, c'') \quad \texttt{pts}(\texttt{ret}^{m''}, c'') \subseteq \texttt{pts}(x, c) \end{array}} \; [\textsc{Call}]$$

Fig. 4. Rules for $k$obj with support for context debloating. m is the method containing the statements analyzed.

finally, (7) `paramsAndRet` : $\mathbb{M} \hookrightarrow \mathcal{P}(\mathbb{V})$ returns the set of the parameters and return variable of a method. Additional notations and functions will be introduced as required.

3.2 Object-Sensitive Pointer Analysis with Support for Context Debloating

Figure 4 gives a set of five rules for performing $k$obj on a Java program consisting of five kinds of statements. Three auxiliary functions are used: (1) `contexts` : $\mathbb{M} \hookrightarrow \mathcal{P}(\mathbb{C})$ maintains the contexts used in analyzing a method, (2) `dispatch` : $\mathbb{H} \times \mathbb{L} \hookrightarrow \mathbb{M}$ resolves an instance call to a target method, and (3) `pts` : $(\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C} \hookrightarrow \mathcal{P}(\mathbb{H} \times \mathbb{C})$ records the points-to information found context-sensitively for a variable or an object's field. All five rules except the premise enclosed in the dashed box of [New] are standard. Note that the "`this`" variable in [Call] is handled differently from the other parameters and $c'' \in \texttt{contexts}(m'')$ in the conclusion of [Call] reveals how the contexts of a method are maintained. Static methods are analyzed using the contexts of their closest callers, which are instance methods (on the call stack) [He et al. 2022; Smaragdakis et al. 2011].

In [New], c' is known as the *heap context* of O. The premise enclosed in the dashed box is specific to context debloating. Here, $\mathcal{I}$ denotes the set of context-independent objects identified by a context debloating approach such as Conch [He et al. 2021a, 2023a] or DebloaterX, which will be introduced in Section 4. Hence, the objects in $\mathbb{H} \setminus \mathcal{I}$ are considered context-dependent objects. For a context-independent object in $\mathcal{I}$, we instruct $k$obj to analyze it context-insensitively by setting its heap context as $c' = [\,]$. This eliminates the context explosion problem that would have occurred when the object is used as a receiver object of an invoked method. For a context-dependent object, we apply the same truncation rule as $k$obj by reducing the context length to $k - 1$. Thus, the traditional $k$obj [Milanova et al. 2002, 2005] is a special case of this formalization where $\mathcal{I} = \emptyset$.

4 DEBLOATERX: A CONTAINER-USAGE-PATTERN-BASED APPROACH

We present DebloaterX, our approach for identifying context-independent objects to enable context debloating for $k$obj. Algorithm 1 outlines the four steps taken by DebloaterX to achieve this goal. Step 1 (Section 4.1) involves encoding a program P into a graph called the *DebloaterX Pointer Assignment Graph* (`XPAG`). In Step 2 (Section 4.2), DebloaterX finds a set of container objects, denoted by `containers`, in `XPAG`. This involves computing `openTypes`, a set of "abstract" types defined precisely below, as well as `inParams` and `outParamsRets`, two mappings from an

Algorithm 1: DebloaterX: finding context-independent objects for context debloating.

```
Input   : $P$ (Input Program)
Output  : $\mathcal{I}$ (Context-Independent Objects)
begin
    $\boxed{\text{Step 1:}}$ Build XPAG (Figure 5).
    $\boxed{\text{Step 2:}}$ Find container objects in $P$.
        $\boxed{\text{Step 2.1:}}$ Compute openTypes (Figure 6).
        $\boxed{\text{Step 2.2:}}$ Compute inParams and outParamsRets (Algorithm 2).
        $\boxed{\text{Step 2.3:}}$ Collect containers (Figure 8).
    $\boxed{\text{Step 3:}}$ Find context-dependent objects according to container-usage patterns.
        $\mathcal{D} \leftarrow \emptyset$
        foreach object $o \in$ containers do
            $\boxed{\text{Step 3.1:}}$ Pattern 1: Inner containers.
            if isAnInnerContainer($o$) then $\mathcal{D} \ni o$ ;
            $\boxed{\text{Step 3.2:}}$ Pattern 2: Factory-Created Containers.
            if isAFactoryCreatedContainer($o$) then $\mathcal{D} \ni o$ ;
            $\boxed{\text{Step 3.3:}}$ Pattern 3: Container Wrappers.
            if isAContainerWrapper($o$) then $\mathcal{D} \ni o$ ;
    $\boxed{\text{Step 4:}}$ Return the set of context-independent objects for context debloating
        foreach object $o \notin \mathcal{D}$ do $\mathcal{I} \ni o$;
        return $\mathcal{I}$
```

object field `f` to the set of parameters whose values may be stored into `this.f.*` and the set of parameters and return variables whose values may be retrieved from `this.f.*`, respectively. Here, `this` represents the *this* variable of some method and `*` in `this.f.*` denotes a sequence of zero or more fields. Based on these analysis results, DebloaterX then determines whether an object is a container or not. In Step 3 (Section 4.3), DebloaterX identifies the usage patterns of container objects and determines which ones are context-dependent. Finally, in Step 4, DebloaterX simply returns the set of context-independent objects, denoted by $\mathcal{I}$, whose contexts will be debloated.

4.1 Step 1: Building the DebloaterX Pointer Assignment Graph (XPAG)

Given a program, its XPAG representation is a form of pointer assignment graph (PAG) [Lhoták and Hendren 2003]. This graph encodes how objects flow in the program and is specifically tailored for our analysis purposes. Figure 5 gives the rules for building XPAG. We have designed XPAG this way since it allows us to accurately and efficiently identify context-independent objects in P.

Rules [X-New], [X-Assign], [X-Load], and [X-Store] handle new statements, assignment statements, load and store statements, respectively, and are standard as in [He et al. 2021a; Lu et al. 2021a; Sridharan and Bodík 2006]. For instance calls, we use [X-Special] and [X-Virtual] to handle special calls (i.e., invocations to private methods and constructors via `invokespecial` in the JVM) and virtual/interface calls (i.e., invocations to virtual methods/interface methods via `invokevirtual/invokeinterface`), respectively. In [X-Special], the parameter passing is modeled as assign (as in [Sridharan and Bodík 2006]) because the unique calling target for a special call is statically known. In [X-Virtual], the parameter passing is modeled as cstore and cload, similarly to [He et al. 2021b], but with no parameter-specific fields being introduced. Therefore, we assume

$$\frac{x = \text{new T} \ // \ O}{O \xrightarrow{\text{new}} x}\ [\text{X-New}] \qquad \frac{x = y}{y \xrightarrow{\text{assign}} x}\ [\text{X-Assign}]$$

$$\frac{}{p_i^m \xleftarrow{\text{param}} p_i^m}\ [\text{X-Param}] \qquad \frac{}{\texttt{ret}^m \xrightarrow{\text{return}} \texttt{ret}^m}\ [\text{X-Return}]$$

$$\frac{x = y.f}{y \xrightarrow{\text{load[f]}} x}\ [\text{X-Load}] \qquad \frac{x.f = y}{y \xrightarrow{\text{store[f]}} x}\ [\text{X-Store}]$$

$$\frac{m \text{ is an instance method}}{\texttt{this}^m \xrightarrow{\text{this}} \text{R} \quad \texttt{this}^m \xrightarrow{\text{param}} \texttt{this}^m}\ [\text{X-This}]$$

$$\frac{l : x = y.m'(a_1, ..., a_n) \text{ is a special call}}{y \xrightarrow{\text{assign}} \texttt{this}^{m'} \quad \texttt{ret}^{m'} \xrightarrow{\text{assign}} x \quad \text{for } 1 \le i \le n,\ a_i \xrightarrow{\text{assign}} p_i^{m'}}\ [\text{X-Special}]$$

$$\frac{l : x = a_0.m'(a_1, ..., a_n) \text{ is a virtual call or an interface call}}{\text{for } 0 \le i \le n,\ a_i \xrightarrow{\text{cstore}} a_0 \quad a_0 \xrightarrow{\text{cload}} x}\ [\text{X-Virtual}]$$

Fig. 5. Rules for constructing the XPAG. Rules for static calls are similar to [X-Special] and elided.

that an argument (a returned value) at a virtual/interface call is stored into (loaded from) any field of its receiver variable. That is, $a_i \xrightarrow{\text{cstore}} a_0$ implies $a_0.* = a_i$ and $a_0 \xrightarrow{\text{cload}} x$ implies $x = a_0.*$. The over-approximation is required for efficiently computing `inParams` and `outParamsRets` in Step 2.2 of Algorithm 1 without determining the calling targets for virtual/interface calls. This approximation enables an estimated calculation of what flows into a method via its parameters and what may flow out of it via its parameters or return variable. Otherwise, tracking such an interprocedural value flow more precisely by using the calling targets computed by a context-insensitive pointer analysis like Spark [Lhoták and Hendren 2003] would significantly increase DebloaterX's analysis time, making it unsuitable as a pre-analysis. It is important to mention that static calls are handled in a similar manner to [X-Special], which significantly contributes to the effectiveness of DebloaterX. Without this handling, if special and static calls were processed using [X-VIRTUAL] as described in [He et al. 2021b], Algorithm 2 would become a fully intra-procedural analysis. This would lead to over-approximated computations of `inParams` and `outParamsRets` in $\boxed{\text{Step 2.2}}$. In addition, [X-Param], [X-Return], and [X-This] add self-loop edges to mark nodes as parameters, return variables, and `this` variables, respectively. In [X-This], the special node R connects all `this` variables in instance methods. DebloaterX analyzes object fields individually, so adding R is merely for convenience and does not affect the analysis's precision, as is clear later.

As XPAG is a form of PAG [Lhoták and Hendren 2003], all edges have their corresponding inverse edges, too. For a regular edge, $x \xrightarrow{l} y$, its inverse is $y \xrightarrow{\bar{l}} x$, which is omitted in Figure 5.

4.2 Step 2: Finding Container Objects

We define container objects by building upon the concept of open types, which are essentially a version of abstract types to be introduced below in Section 4.2.1 for debloating contexts in *k*obj.

Definition 1 (**Container objects**). *An object O is considered a* container object *if it has at least one pointer field f of an open type that receives an incoming value flow in some method m such that*

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

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

Fig. 6. Rules for defining open types.

$\mathtt{this}^m.f.* = p$*, where* p *is a parameter of* m*, and returns an outgoing value flow in some method* m' *such that* $v.* = \mathtt{this}^{m'}.f.*$*, where* v *is either the return variable or a parameter of* m'*. Both* m *and* m' *are methods invoked on* O *(i.e., in* $\mathtt{methodsInvokedOn}(O)$*), and* m' *may or may not be the same as* m*.*

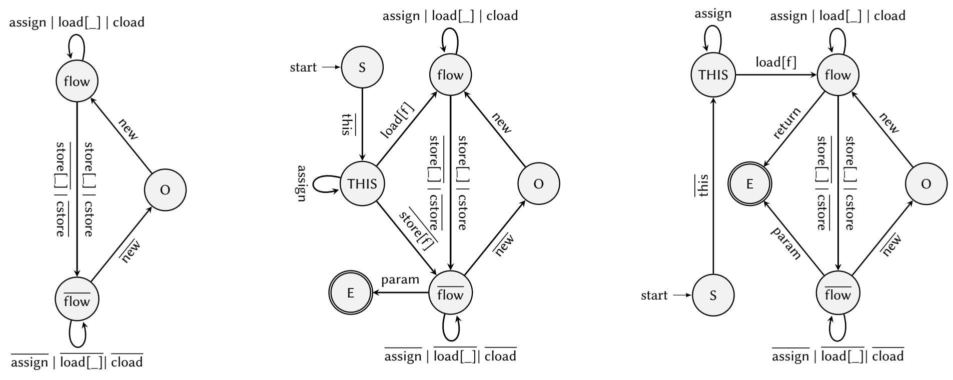
According to this definition, an object O is considered a container object if it has a pointer field of an open type (Section 4.2.1) whose value is both received and returned by methods invoked on O. However, precisely identifying container objects in a program requires some degree of field sensitivity, which can be prohibitively time-consuming. To address this challenge, we use finite-state automata that support 1-limited field-sensitivity and enable parallelism to analyze each object field individually and reason about its incoming and outgoing value flows (Section 4.2.2). In Section 4.2.3, we introduce the rules for determining whether an object is a container object.

4.2.1 Open Types. A type is considered an *open type* if it is either an abstract type itself (including `java.lang.Object`) or contains at least one field of an abstract type. In Definition 1, we require a container object to have at least one field of an open type. This definition is based on our practical experience and observations rather than any theoretical guarantees. It is worth noting that modern Java pointer analysis frameworks, such as Qilin [He et al. 2022], Doop [Bravenboer and Smaragdakis 2009], and Wala [WALA 2023], all support type filtering, which prevents type-incompatible objects from being assigned to a pointer variable. For a program that only contains objects with non-open-type fields, moving from context insensitivity to object sensitivity does not lead to a significant improvement in precision. This is because, in real-world applications, when dealing with a pointer variable of a non-open type, the object-sensitive pointer analysis typically filters out numerous spurious objects that end up in its points-to set. Our evaluation in Section 5 confirms the effectiveness of this filtering process, as our context-debloating approach demonstrates nearly precision-preserving results for a set of commonly used metrics. However, there are rare cases, highlighted in Figure 17, where solely relying on the filtering process may not be sufficient.

Figure 6 gives the five rules for defining open types. According to the three rules on the top, the `java.lang.Object` type, as well as abstract types and interface types, are considered open types. In addition, if a type t is an open type, its array form $[t]$ is also an open type. Note that an array O of type $[t]$ is modeled to have a special field `arr` of type t such that O.`arr` stores its elements. Finally, if a field f declared in t or any supertype of t (without being overridden) is an open type, then t is considered an open type. All open types in a program are collected in the set openTypes.

Example 4.1. *In Figure 1, the types* `[Object]` *and* `Entry` *are open types because they contain a field of the* `java.lang.Object` *type. Similarly,* `[Entry]`, `HashMap`, `HashSet`, `KeyIterator`*, and* `KeySet` *are also considered open types because each of them contains at least one open-type field.*

4.2.2 Incoming and Outgoing Value Flow Analyses. To determine if an object is a container object as specified in Definition 1 according to the rules in Figure 8, we conduct incoming and outgoing value flow analyses to track the flow of values through $\mathtt{this}^m.\mathtt{f}.*$ starting or ending at parameters and return variables, for every method m. To ensure both precision and efficiency, DebloaterX processes each object field in the program individually, supporting 1-limited field sensitivity and



(a) $\mathcal{A}_{comm}$ (b) $\mathcal{A}_{in}^{f}$ (c) $\mathcal{A}_{out}^{f}$

Fig. 7. DFAs for performing the incoming and outgoing value flow analyses for a field f. $\mathcal{A}_{in}^{f}$ checks whether the objects stored in `this.f.*` can originate from any parameter and $\mathcal{A}_{out}^{f}$ checks whether the objects stored in `this.f.*` can be returned by any return variable or parameter. $\mathcal{A}_{comm}$ represents their common part.

allowing for parallelism. For each object field f, we determine the set of parameters (of all methods in the program) whose values may flow into $\texttt{this}^m\texttt{.f.*}$ in the *incoming value-flow analysis*, and the set of parameters or return variables (of all methods in the program) that may store the values recorded in $\texttt{this}^m\texttt{.f.*}$ in the *outgoing value-flow analysis*, for every method m. To accomplish this goal, DebloaterX has designed two deterministic finite automata (DFAs) for each field f: $\mathcal{A}_{in}^{f}$ (Figure 7b) for the incoming analysis and $\mathcal{A}_{out}^{f}$ (Figure 7c) for the outgoing analysis.

We begin by introducing a separate DFA, referred to as $\mathcal{A}_{comm}$, which is a shared component of both $\mathcal{A}_{in}^{f}$ and $\mathcal{A}_{out}^{f}$ and is illustrated in Figure 7a. To construct $\mathcal{A}_{comm}$, we utilize a standard CFL-reachability formulation (i.e., L_{FT} in [Sridharan et al. 2005]) and regularize it through a series of similar approximation steps as outlined in [He et al. 2021b], using a standard regularization approximation algorithm for context-free grammar [Mohri and Nederhof 2001]. It is worth noting that $\mathcal{A}_{comm}$ is intentionally designed to over-approximate both the incoming and outgoing value-flows of object fields, including the related aliases, in XPAG for the specific purpose of determining whether an object is a container or not based on Definition 1.

Let us take a closer look at $\mathcal{A}_{in}^{f}$, which is responsible for performing the incoming value-flow analysis for field f. Given an edge $R \xrightarrow{\overline{\text{this}}} \texttt{this}^m$ for some method m in XPAG, this DFA starts at its start state S and then transits to state THIS, which has two non-self-loop outgoing transitions. If a load[f] edge is encountered, the DFA enters state flow to check which variables retrieve the objects stored in $\texttt{this}^m\texttt{.f.*}$ and which parameters can have their values stored into the objects pointed by these variables. If a $\overline{\text{store[f]}}$ edge is encountered, the DFA enters state $\overline{\text{flow}}$ to check if a value stored in $\texttt{this}^m\texttt{.f.*}$ comes from any parameter. If $\mathcal{A}_{in}^{f}$ eventually reaches the final state E via a param edge, then we have found a parameter whose value can be stored into $\texttt{this}^m\texttt{.f.*}$.

$\mathcal{A}_{out}^{f}$ works similarly to $\mathcal{A}_{in}^{f}$, but with a focus on performing the outgoing value-flow analysis for field f. Given an edge $R \xrightarrow{\overline{\text{this}}} \texttt{this}^m$ in XPAG, $\mathcal{A}_{out}^{f}$ starts at its start state S and can only transit to state flow via a load[f] edge. It then checks to see whether the objects stored in $\texttt{this}^m\texttt{.f.*}$ can flow to any return variable or any parameter of some method in the program.

Algorithm 2: Computing `inParams` and `outParamsRets`.

```
Input  : Program $P$ and its XPAG
Output: inParams, outParamsRets
1  foreach field $f$ appearing in $P$ do
2      inParams($f$) = runDFAandCollect($\mathcal{A}^f_{in}$)
3      outParamsRets($f$) = runDFAandCollect($\mathcal{A}^f_{out}$)
4  function nextNodeStates($\mathcal{A}$, node, state):
5      nexts ← ∅, let $\delta$ be the transition function of $\mathcal{A}$
6      foreach $node \xrightarrow{edgekind} node'$ in XPAG do
7          $state' = \delta(state, edgekind)$
8          if $state' \neq$ Err then $nexts \ni \langle node', state' \rangle$ ;
9      return nexts
10 function runDFAandCollect($\mathcal{A}$):
11     W ← {⟨R, S⟩} , $r$ ← visit ← ∅
12     while W ≠ ∅ do
13         $\langle n, s \rangle$ ← poll(W)
14         if $s$ = E then $r \ni n$;
15         for $\langle n', s' \rangle \in$ nextNodeStates($\mathcal{A}$, $n$, $s$) do
16             if $\langle n', s' \rangle \notin$ visit then
17                 W ∋ $\langle n', s' \rangle$, visit ∋ $\langle n', s' \rangle$
18     return $r$
```

As is clear in Figures 7b and 7c, each DFA only supports 1-limited field-sensitivity for the *this* variable of a method. Despite this limitation, DebloaterX is currently the only pre-analysis that provides a certain degree of field sensitivity for improving the effectiveness of *k*obj, resulting in the best precision and efficiency trade-offs, as demonstrated in Section 5.

We have devised an algorithm that efficiently computes `inParams` and `outParamsRets` in Step 2.2 of Algorithm 1. As shown in lines 2-3 of Algorithm 2, we use `runDFAandCollect`($\mathcal{A}^f_{in}$) (`runDFAandCollect`($\mathcal{A}^f_{out}$)) to compute `inParams`(f) (`outParamsRets`(f)) for each object field f in a program P. Here, `runDFAandCollect` (lines 10-18) is a standard worklist algorithm that traverses all reachable node-state pairs starting from ⟨R, S⟩ (where R is the special node introduced in Figure 5). Whenever a node $n \in$ XPAG is reachable under the final state E, n is added to `inParams`(f) or `outParamsRets`(f) (depending on which DFA is used). The `nextNodeStates` function (lines 4-9) simply returns the set of node-state pairs reached from a given node-state pair, as is standard.

Example 4.2. *For the example given in Figure 1, there are seven open-type fields:* `map`, `table`, `keySet`, `m1`, `m2`, `key`, *and* `value`. *All instance calls (except those to constructors) are modeled as virtual calls by [X-Virtual]. According to Algorithm 2, their* `inParams` *sets are respectively* {p}, {q1, q2, p1, p2}, $\{\texttt{this}^{\texttt{keySet}}\}$, $\{\texttt{m3}, \texttt{this}^{\texttt{KeySet:iterator}}\}$, $\{\texttt{m4}, \texttt{this}^{\texttt{keySet}}\}$, {p1, q1}, *and* {p2, q2}, *and their* `outParamsRets` *sets are respectively* $\{\texttt{ret}^{\texttt{HashSet:iterator}}, \texttt{p}\}$, {q1, q2}, $\{\texttt{ret}^{\texttt{keySet}}\}$, $\{\texttt{ret}^{\texttt{next}}\}$, $\{\texttt{ret}^{\texttt{KeySet:iterator}}\}$, ∅, *and* ∅. *The state transition path below explains why* q1 ∈ `inParams(table)` *(computed by Algorithm 2 in terms of its corresponding DFA* $\mathcal{A}^{\texttt{table}}_{in}$*), which is the most complex transition path for all seven fields:*

$$\begin{aligned}&\langle \mathsf{R}, \mathsf{S} \rangle \xrightarrow{\overline{this}} \langle \texttt{this}^{\texttt{put}}, \mathsf{THIS} \rangle \xrightarrow{load[\texttt{table}]} \langle t1, \mathsf{flow} \rangle \xrightarrow{\overline{store[\_]}} \langle t_2, \overline{\mathsf{flow}} \rangle \xrightarrow{\overline{new}} \langle \mathsf{E}, \mathsf{O} \rangle \xrightarrow{new} \\ &\langle t_2, \mathsf{flow} \rangle \xrightarrow{assign} \langle \texttt{this}^{\texttt{Entry}}, \mathsf{flow} \rangle \xrightarrow{\overline{store[\_]}} \langle \texttt{p1}, \overline{\mathsf{flow}} \rangle \xrightarrow{\overline{assign}} \langle \texttt{q1}, \overline{\mathsf{flow}} \rangle \xrightarrow{param} \langle \texttt{q1}, \mathsf{E} \rangle\end{aligned} \tag{1}$$

where `t1` *and* `t2` *are two temporary variables introduced for transforming line 53 into* `t1 = this.table`, `t2 = new Entry(q1, q2)`, *and* `t1[0] = t2` *(as three address statements are required in Figure 5).*

Note that `p1` *and* `p2` *in* `inParams(table)` *are not parameters of any methods declared in* `HashMap`. *So they do not affect how container objects are identified as shown in Figure 8, according to Definition 1.*

4.2.3 Collecting Container Objects. We are now introducing the rules in Figure 8 for identifying container objects in a program. The two rules in [Con] are used for this purpose. For the left rule, an object O is considered a container object if it satisfies Definition 1. That is, O must contain an open-type field f with both an incoming and outgoing value flow (happening when `hasInFlow` (O, f) and

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

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

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

Fig. 8. Rules for defining container objects, where '_' is a placeholder whose content is not of our concern.

hasOutFlow (O, f) hold, respectively). For the right rule, any array object used to store elements of an open type is always considered a container object, without having to verify hasInFlow (O, f) and hasOutFlow (O, f). This is because these two kinds of value flow typically exist in practice.

Given a node $n \in$ XPAG of a program, we define $assign^*(n)$ to be the set of all reachable nodes in XPAG from n by traversing only the assign edges (i.e., the edges labeled with assign). In all these rules, $\overline{\texttt{pts}}(v)$ gives the points-to set computed by a context-insensitive pointer analysis.

In [In], we provide two rules for checking whether an object O has an incoming value flow on field f. For the left rule, we apply the corresponding constraint specified in Definition 1 directly. For the right rule, we assume that $\texttt{hasInFlow}(O, f)$ holds conservatively, based on a crucial observation made in applying $k$obj to analyze many Java codebases. More specifically, if there exists a store statement $a.f = \ldots$ in method m, where $O \in \overline{\texttt{pts}}(a)$ is not allocated in m, such that $a \notin assign^*(\texttt{this}^m)$ (otherwise, the left rule may be applicable), then we assume that $\texttt{hasInFlow}(O, f)$ holds.

Similarly, in [Out], we provide two rules for checking whether an object O has an outgoing value flow on field f. For the left rule, we apply the corresponding constraint in Definition 1 directly. For the right rule, we assume that $\texttt{hasOutFlow}(O, f)$ holds when checking the existence of an outgoing value flow for $O.f$ due to a similar observation as discussed above for the right rule of [In].

The left rule in [In] ([Out]) is more commonly used than the right rule in object-oriented languages, accounting for an average of 91.7% of container objects found in our evaluation.

Example 4.3. *In Figure 1,* O1, O2, O3, C1, *and* C2 *are not containers because they do not have any instance fields.* [O] *and* [E] *are containers because they are open-type arrays.* S *is a container as it satisfies the following three conditions: (1) its field* map *is of an open type, (2)* hasInFlow (S, map) *holds due to* add() *being invoked on* S *and the parameter* p ∈ inParams(map)*, (3)* hasOutFlow (S, map) *holds due to* HashSet:iterator() *being invoked on* S *and* $\texttt{ret}^{\texttt{HashSet:iterator}}$ ∈ outParamsRets(map)*. For* M, KS, *and* KI, *we similarly conclude that they are containers. Note that* hasOutFlow (M, table) *holds due to both* q1 ∈ outParamsRets(table) *and the load to* table *in line 59. The latter is expected while the former is a false positive due to the limitation of our automata (which over-approximates value flow by using, for example, 1-limited field sensitivity).* E *is a container due to (1) the field* key *is of an open type, (2)* hasInFlow (E, key) *holds since* p1 ∈ inParams(key)*, and (3)* hasOutFlow (E, key) *holds since there is a load to* key *in line 59 of Figure 1 where the load base is not a this variable.*

4.3 Step 3: Finding Context-Dependent Objects Based on Container-Usage Patterns

We now provide efficient rules and algorithms for identifying three container-usage patterns.

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

Fig. 9. Rule for identifying inner containers.

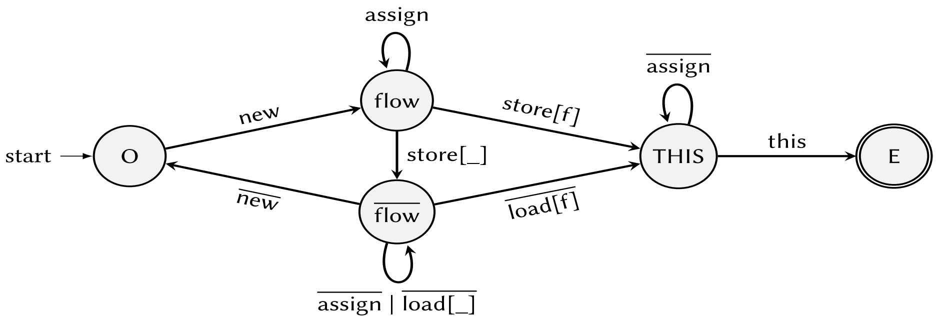


Fig. 10. An NFA for checking whether an object O can be stored into `this.f.*`.

4.3.1 Pattern 1: Inner Containers. Figure 9 provides a rule for identifying inner containers. A container object O allocated in an instance method m is considered an *inner container* if there exists a receiver object $O' \in \texttt{receiverObjects}(m)$ satisfying that (1) O can be stored into $O'.f.*$ via $\texttt{this}^m.f.*$ (due to $f \in \texttt{objectStoredInto}(O)$) and (2) O' is a container object as per Definition 1 (due to $t = \texttt{typeof}(f)$, $t \in \texttt{openTypes}$, $\texttt{hasInFlow}(O', f)$, and $\texttt{hasOutFlow}(O', f)$). Here, `objectStoredInto` is a mapping from an object o' allocated in method m' to the set of fields f' such that o' can be stored into $\texttt{this}^{m'}.f'.*$. Its construction is explained below.

We begin with an NFA (Non-Deterministic Finite Automaton), depicted in Figure 10, to determine whether an object O can be stored into `this.f.*`, where `this` can refer to the *this* variable of any method being analyzed. We then refine these analysis results by constructing `objectStoredInto`.

The NFA starts with an object $O \in$ XPAG from its start state O and transits to state flow by following a sequence of XPAG edges (starting with a new edge) until reaching a node $x \in$ XPAG. Suppose a store edge $x \xrightarrow{\text{store[f]}} y$ exists in XPAG. If the NFA can eventually transit to the final state E, then we know that O has been stored into some `this.f.*` (due to [X-This] in Figure 5). Otherwise, the NFA can also transit to state $\overline{\text{flow}}$. If the NFA can eventually transit from $\overline{\text{flow}}$ to state THIS via an $\overline{\text{load[f]}}$ edge and then reach the final state E, we can conclude that O has been stored into some `this.f.*`. However, if the NFA reaches a node O' by transitting from $\overline{\text{flow}}$ to state O such that $O'.* = O$, it will check whether O' can be stored into some `this.f.*`. To prevent false positives, we have constructed the NFA to disregard all cstore and cload edges (and their inverses) in XPAG. This balance between efficiency and precision has been extensively tested and proven effective, with only a negligible loss of precision in rare real-world scenarios, as demonstrated in Figure 17c.

Based on this NFA, we can construct `objectStoredInto`, as shown in Algorithm 3, by ensuring that if $f \in \texttt{objectStoredInto}(O)$ (lines 18-19), then O may be stored into $\texttt{this}^m.f.*$, where $\texttt{this}^m$ is the *this* variable of the method m in which O is allocated (lines 5 and 17).

4.3.2 Pattern 2: Factory-Created Containers. As shown in Figure 11, identifying factory-created containers is straightforward. An object O allocated in method m is a *factory-created container* if m is a static method and O can be directly returned in m (i.e., when $\texttt{isDirectlyReturned}(O)$ holds).

Algorithm 3: Computing `objectStoredInto`.

```
Input  : Program $P$, its XPAG, and the NFA $\mathcal{N}$ in Figure 10
Output : objectStoredInto
foreach object $O$ allocated in $P$ do
    objectStoredInto($O$) = runNFAandCollect($O$)
function runNFAandCollect($O$):
    $W \leftarrow \{\langle O, S\rangle\}$ , $r \leftarrow visit \leftarrow \emptyset$, $m \leftarrow$ methodof($O$),
    $thisAlias \leftarrow assign^*(\mathtt{this}^m)$
    while $W \neq \emptyset$ do
        $\langle n, s\rangle \leftarrow$ poll(W)
        for $\langle n', s'\rangle \in$ nextNodeStates($thisAlias, n, s, r$) do
            if $\langle n', s'\rangle \notin visit$ then $W \ni \langle n', s'\rangle$, $visit \ni \langle n', s'\rangle$;
    return $r$
function nextNodeStates($thisAlias, node, state, r$):
    $nexts \leftarrow \emptyset$, let $\delta$ be the transition function of $\mathcal{N}$
    foreach $node \xrightarrow{edgekind} node'$ in XPAG do
        $nextStates = \delta(state, edgekind)$
        foreach $state' \in nextStates$ do
            if $state' \neq$ Err then $nexts \ni \langle node', state'\rangle$ ;
            if $state'$ = THIS and $node' \in thisAlias$ then
                assert $edgekind$ = store[f] or $\overline{\text{load[f]}}$
                $r \ni$ f
    return $nexts$
```

$$\frac{m \text{ is a static method} \qquad m = \mathtt{methodof}(O) \qquad \mathtt{isDirectlyReturned(O)}}{\mathtt{isAFactoryCreatedContainer}(O)} \qquad \frac{O \xrightarrow{\text{new}} x \text{ is an edge in } \mathtt{XPAG} \qquad m = \mathtt{methodof}(O) \qquad \mathtt{ret}^m \in assign^*(x)}{\mathtt{isDirectlyReturned}(O)}$$

Fig. 11. Rules for identifying factory-created containers.

4.3.3 Pattern 3: Container Wrappers. In Figure 12, we give the rules for identifying container wrappers. A container object O is considered a *container wrapper* if it is allocated in an instance method m, directly returned in m (i.e., when `isDirectlyReturned`(O) holds), and contains a field whose content comes from a parameter of m (i.e., when `isContentFromParam` (O) holds).

Given a node $n \in$ XPAG, we define $load^*(n)$ to be the set of all reachable nodes in XPAG from n by traversing only assign, load[_], and cload edges. We define `isContentFromParam` by distinguishing two cases. If O is an array object, `isContentFromParam` (O) holds if there exists a store edge $y \xrightarrow{\text{store[\_]}} b$ in XPAG and a parameter p of m such that $y \in load^*(p)$ (i.e., $y = p.*$) and $b \in assign^*(x)$ (i.e., the value of b comes directly from x), where x = new O. On the other hand, if O is a non-array object, `isContentFromParam` (O) holds if three conditions are satisfied: (1) O is a container object (due to $t' = \mathtt{typeof}(f)$, $t' \in \mathtt{openTypes}$, $\mathtt{hasInFlow}(O, f)$, $\mathtt{hasOutFlow}(O, f)$), (2) the value of f comes from a parameter $p_i^{m''}$ (i.e., $p_i^{m''} \in \mathtt{inParams}(f)$), where method m'' is invoked at a call site $l : b.m'(a_1, \cdots, a_n)$ in m with the value of b directly coming from O (i.e., $b \in assign^*(x)$ and x = new O), and (3) $a_i \in load^*(p)$, where p is a parameter of m.

Example 4.4. *For the example in Figure 1, the identified inner containers include* `M`, `[E]`, `KS`, *and* `E`, *while the container wrappers include* `[O]`, `KS`, *and* `KI`. *Only* `S` *is a factory-created container, and all the other objects are selected to be context-independent. Here, we use* `KS` *as a representative to explain our rules. First, since* `keySet` ∈ `objectStoredInto(KS)` *(due to line 48),* `M` *is a receiver object of* `keySet()`, *and both* `hasInFlow` *(*`M, keySet`*) and* `hasOutFlow` *(*`M, keySet`*) hold, all premises in Figure 9 are satisfied, making* `KS` *an inner container. Second, since* `KS` *can be directly returned in* `keySet()` *and* $\mathtt{this}^{\mathtt{keySet}}$ *can be saved into* `KS.m2` *via line 47, according to the rules in Figure 12,* `KS` *is also a container wrapper. As shown in Figure 13 in our experimental evaluation, the overlap between inner containers and container wrappers is generally small in real-world Java programs.*

4.4 Soundness and Complexity Analysis

We discuss the soundness as well as the time and space complexities of DebloaterX.

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

$$\frac{\begin{array}{c} O \xrightarrow{\text{new}} x \text{ is an edge in } \texttt{XPAG} \quad b \in \mathit{assign}^*(x) \quad t = \texttt{typeof}(O) \quad \text{t is an array type} \\ m = \texttt{methodof}(O) \quad p \in \texttt{params}(m) \quad y \in \mathit{load}^*(p) \quad y \xrightarrow{\text{store[\_]}} b \text{ is an edge in } \texttt{XPAG} \end{array}}{\texttt{isContentFromParam}(O)}$$

$$\frac{\begin{array}{c} O \xrightarrow{\text{new}} x \text{ is an edge in } \texttt{XPAG} \quad b \in \mathit{assign}^*(x) \quad t = \texttt{typeof}(O) \quad \text{t is a non-array type} \\ m = \texttt{methodof}(O) \quad p \in \texttt{params}(m) \quad l : b.m'(a_1, \cdots, a_n) \text{ is a call in } m \\ a_i \in \mathit{load}^*(p) \quad f \in \mathbb{F} \quad t' = \texttt{typeof}(f) \quad t' \in \texttt{openTypes} \\ m'' = \texttt{dispatch}(O, l) \quad p_i^{m''} \in \texttt{inParams}(f) \quad \texttt{hasInFlow}(O, f) \quad \texttt{hasOutFlow}(O, f) \end{array}}{\texttt{isContentFromParam}(O)}$$

Fig. 12. Rules for identifying container wrappers.

Soundness. Let $\mathcal{I}$ in [New] of Figure 4 be computed by DebloaterX using Algorithm 1. No matter what $\mathcal{I}$ is, $k$obj remains sound, because the objects in $\mathcal{I}$ will be analyzed context-insensitively after debloating, ensuring soundness, albeit with possible imprecision.

Complexity Analysis. Table 1 shows the time and space complexities of each step in DebloaterX. The overall time and space complexities are $O((|\mathbb{F}|+|\mathbb{H}| \cdot \alpha_p) \cdot |\mathbb{L}|) + |\mathbb{H}| \cdot |\mathbb{F}_t| \cdot |\mathbb{H}_r|)$ and $O(\alpha_p \cdot (|\mathbb{L}| + |\mathbb{F}| \cdot |\mathbb{M}|))$, respectively. For Java, α_p, and $|\mathbb{F}_t|$ are considered as constants, and we have $\mathbb{H}_r \leq \mathbb{H} \leq \mathbb{L}$. Thus, the complexities can be simplified to $O((|\mathbb{F}| + |\mathbb{H}|) \cdot |\mathbb{L}|)$ and $O(|\mathbb{L}| + |\mathbb{F}| \cdot |\mathbb{M}|)$. All the rules and algorithms given in Algorithm 2 and Figures 8, 9, 11 and 12 can run concurrently, reducing the time complexity to $O(|\mathbb{L}|)$ if $O(|\mathbb{F}| + |\mathbb{H}|)$ CPU cores are available. As the size of XPAG is $O(\mathbb{L})$, the time complexity of DebloaterX in the concurrent setting is linear to the size of XPAG.

Table 1. Time and space complexities of DebloaterX. Here, α_p is the maximum number of parameters of a method, and $|\mathbb{F}_t|$ is the maximum number of fields that can be accessed by an instance object of a type t, and $|\mathbb{H}_r|$ is the maximum number of receiver objects of a method in a program.

Steps		Time Complexity	Space Complexity
Step 1	Figure 5	$O(\|\mathbb{L}\| \cdot \alpha_p + \|\mathbb{M}\| \cdot \alpha_p)$	$O(\|\mathbb{L}\| \cdot \alpha_p + \|\mathbb{M}\| \cdot \alpha_p)$
Step 2	Figure 6	$O(\|\mathbb{T}\| \cdot \|\mathbb{F}_t\|)$	$O(\|\mathbb{T}\|)$
	Algorithm 2	$O(\|\mathbb{F}\| \cdot \|\mathbb{L}\|)$	$O(\|\mathbb{F}\| \cdot \alpha_p \cdot \|\mathbb{M}\| + \|\mathbb{L}\|)$
	Figure 8	$O(\|\mathbb{H}\| \cdot \|\mathbb{F}_t\|)$	$O(\|\mathbb{H}\|)$
Step 3	Figure 9	$O(\|\mathbb{H}\| \cdot \|\mathbb{F}_t\| \cdot \|\mathbb{H}_r\|)$	$O(\|\mathbb{H}\|)$
	Algorithm 3	$O(\|\mathbb{H}\| \cdot \|\mathbb{L}\|)$	$O(\|\mathbb{L}\| + \|\mathbb{F}_t\|)$
	Figure 11	$O(\|\mathbb{H}\| \cdot \|\mathbb{L}\|)$	$O(\|\mathbb{H}\| + \|\mathbb{L}\|)$
	Figure 12	$O(\|\mathbb{H}\| \cdot (\|\mathbb{L}\| + \|\mathbb{L}\| \cdot \alpha_p))$	$O(\|\mathbb{H}\| + \|\mathbb{L}\|)$
Step 4		$O(\|\mathbb{H}\|)$	$O(\|\mathbb{H}\|)$

As previously discussed in Section 4.1, we can modify our approach to be implemented purely intra-procedurally. This can be achieved by treating special calls and static calls as [X-Virtual], following a similar methodology as described in [He et al. 2021b] when constructing XPAG according to Figure 5. However, it is important to note that such an implementation would result in significant over-approximation when computing `inParams` and `outParamsRets` in Step 2.2. As a consequence, this implementation would incorrectly classify a larger number of context-independent objects as context-dependent. Consequently, the effectiveness of DebloaterX would be greatly diminished.

5 EVALUATION

We evaluate the effectiveness of our context-debloating approach by addressing four **RQ**s:

- **RQ1.** Does DebloaterX effectively debloat $k$obj by significantly improving its efficiency while maintaining its precision in a robust manner?
- **RQ2.** How does DebloaterX compare to state-of-the-art pre-analysis techniques, such as Zipper [Li et al. 2018] and Conch [He et al. 2021a], in terms of precision and efficiency?
- **RQ3**. How does DebloaterX compare to a simplistic approach that identifies context-dependent objects solely based on whether they are containers, which are essentially determined by their relation to `java.util.Object` and `java.util.Collection`?
- **RQ4**. What causes the slight precision decrease and scalability issues of the debloated $k$obj?

We compare X-$k$obj with Z-$k$obj, C-$k$obj, and S-$k$obj by using $k$obj as the baseline. In this comparison, Z-$k$obj represents the version of $k$obj with selective context sensitivity achieved through Zipper [Li et al. 2018], while X-$k$obj and C-$k$obj are the versions of $k$obj with debloated contexts using DebloaterX and Conch [He et al. 2021a], respectively. Finally, S-$k$obj is the version of $k$obj debloated using a simplistic pre-analysis approach that determines context-dependability based on whether an object is a container type or not, as defined below. According to this approach, a class is considered a container if it has a `java.lang.Object` field or a container field, or if it implements `java.util.Collection` or is nested in a class implementing `java.util.Collection`.

We do not compare X-$k$obj with any kCFA-based pointer analysis and its context reduction techniques (e.g., Zipper-guided kCFA [Li et al. 2020]) since they are known to underperform $k$obj in terms of both precision and efficiency. We also do not compare X-$k$obj with techniques such as context tunnelling [Jeon et al. 2018] that have distinct design objectives with us.

An analysis is *scalable* for a program if it can be completed within a given time/memory budget.

Implementation. We have implemented DebloaterX in Qilin [He et al. 2022], an open-source Java pointer analysis framework that supports fine-grained selective context-sensitivity on top of Soot [Vallée-Rai et al. 2010]. Our implementation is compact, with less than 1500 lines of Java code at its core, thanks to our insightful approach, straightforward rules, and efficient algorithms. To ensure reproducibility, we have released its source code at https://github.com/DongjieHe/DebloaterX and a Docker image at Docker Hub. Additionally, within Qilin, we have implemented the simplistic `java.lang.Collection`-based pre-analysis approach to obtain S-$k$obj in approximately 200 lines of Java code. The other tools used here, such as Spark (a context-insensitive pointer analysis), $k$obj, Conch, and Zipper, are already available in Qilin.

Experimental Setup. We have conducted all experiments on a machine with an Intel(R) Xeon(R) W-2245 3.90GHz CPU and 512GB memory, with a time budget of 12 hours per program. To ensure fairness in comparing Conch and Zipper, we have used the same analysis setting as Conch, which closely matches the one used in Zipper [Li et al. 2018]. Specifically, native code is handled using hard-coded summaries provided in Soot, and Java reflection is resolved using Tamiflex logs [Bodden et al. 2011]. We follow the practice of distinguishing objects instantiated from `StringBuilder`, `StringBuffer`, and `Throwable` (including its subtypes) by their dynamic types, and then analyzing them context-insensitively, as done in Qilin [He et al. 2022], Doop [Bravenboer and Smaragdakis 2009], and Wala [WALA 2023]. If this manual heuristic is disabled, DebloaterX would perform even more impressively, as it would automatically identify such objects as context-independent.

Benchmark Selection. We have selected 12 popular Java programs for our evaluation, including 5 benchmarks (the first five in Table 2) from `DaCapo2006` [Blackburn et al. 2006], 2 real-world Java applications (the middle two in Table 2), and 5 benchmarks (the last five in Table 2) from a more recent version of DaCapo (`DaCapo-9.12`) downloaded from Doop benchmarks. For the `DaCapo2006` benchmarks and real-world applications, we use `jre1.6.0_45`, while for the `DaCapo-9.12` benchmarks, we use `jre1.8.0_121_debug`, a relatively larger and more complex Java library, and the default Java reflection log (with the suffix `-tamiflex-default.log`). Our benchmark selection criteria include the availability of diverse benchmarks to appraise the robustness of DebloaterX, program size (in terms of the number of reachable methods computed by Spark), and a balanced ratio among the three benchmark sources. We have included `checkstyle` because it is the only one that can be analyzed by X-3obj but not by C-3obj. We utilize a subset of the benchmarks from `DaCapo-9.12` because neither 3obj nor Z-3obj can analyze any benchmarks in `DaCapo-9.12` scalably. While this further demonstrates the superiority of DebloaterX in accelerating 3obj, it makes evaluating DebloaterX against 3obj and Z-3obj in terms of precision to be impossible.

Our main results are given in Table 2, which includes the performance of the baseline $k$obj, as well as Z-$k$obj, S-$k$obj, C-$k$obj, and X-$k$obj for each $k \in \{2, 3\}$. We have omitted the results of Eagle, a precision-preserving pre-analysis for $k$obj [Lu and Xue 2019], in Table 2 because (1) the precision of E-$k$obj (the version of $k$obj obtained by Eagle) can be read off from that of $k$obj, and (2) E-$k$obj is the slowest of the four pre-analysis techniques in terms of efficiency.

We evaluate the efficiency of all the pointer analyses considered in terms of analysis time and speedup compared to the baseline $k$obj. For measuring precision, we consider four commonly used metrics [He et al. 2021b; Jeon et al. 2020; Jeon and Oh 2022; Li et al. 2022]: (1) `#fail-casts`, which measures the number of type casts that may fail, (2) `#call-edges`, which counts the number of discovered call graph edges, (3) `#reachables`, which measures the number of reachable methods, and (4) `#poly-calls`, which counts the number of discovered polymorphic calls.

Let $\mathcal{A}^k \in \{\textsc{Spark}, k\text{obj}, Z\text{-}k\text{obj}, S\text{-}k\text{obj}, C\text{-}k\text{obj}, X\text{-}k\text{obj}\}$, where $k \in \{2, 3\}$, denote a pointer analysis evaluated in Table 2, and let $M_{\mathcal{A}^k}$ denote the result of a precision metric M obtained by $\mathcal{A}^k$. We define the precision loss of $\mathcal{A}^k$ with respect to the baseline $k$obj on metric M as:

$$\Delta^M_{\mathcal{A}^k} = \frac{(M_{\textsc{Spark}} - M_{k\text{obj}}) - (M_{\textsc{Spark}} - M_{\mathcal{A}^k})}{M_{\textsc{Spark}} - M_{k\text{obj}}} = \frac{M_{\mathcal{A}^k} - M_{k\text{obj}}}{M_{\textsc{Spark}} - M_{k\text{obj}}} \tag{2}$$

Obviously, we have $\Delta^M_{k\text{obj}} = 0\%$ and $\Delta^M_{\textsc{Spark}} = 100\%$. The precision loss of Z-$k$obj, S-$k$obj, C-$k$obj, and X-$k$obj lies in between, as they are different versions of $k$obj with potential precision loss.

5.1 RQ1: The Precision, Efficiency, and Robustness of X-$k$obj

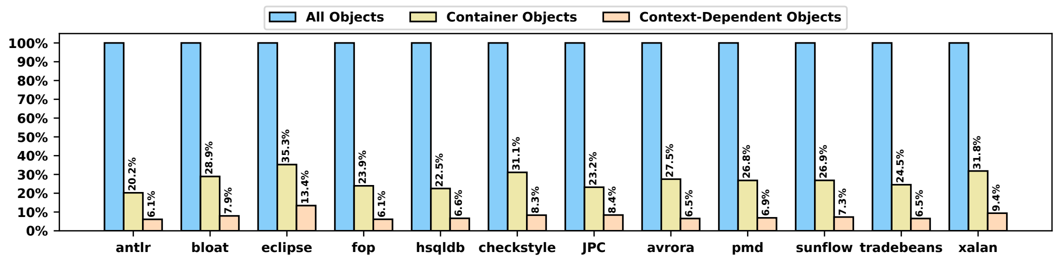
DebloaterX is highly effective as X-$k$obj improves $k$obj significantly in terms of the three criteria.

Precision. DebloaterX achieves nearly identical precision as $k$obj for every metric in every program. When $k = 2$, the average precision losses of X-2obj with respect to 2obj are $\Delta^{\texttt{\#fail-casts}}_{X\text{-2obj}} = 0.2\%$, $\Delta^{\texttt{\#call-edges}}_{X\text{-2obj}} = 0.0\%$, $\Delta^{\texttt{\#reachables}}_{X\text{-2obj}} = 0.2\%$, and $\Delta^{\texttt{\#poly-calls}}_{X\text{-2obj}} = 0.1\%$, with an overall precision loss of only 0.1%. When $k = 3$, 3obj can analyze only four benchmarks (`antlr`, `fop`, `hsqldb`, and `JPC`) scalably, for which the average precision losses of X-3obj with respect to 3obj are $\Delta^{\texttt{\#fail-casts}}_{X\text{-3obj}} = 0.0\%$, $\Delta^{\texttt{\#call-edges}}_{X\text{-3obj}} = 0.1\%$, $\Delta^{\texttt{\#reachables}}_{X\text{-3obj}} = 0.1\%$, and $\Delta^{\texttt{\#poly-calls}}_{X\text{-3obj}} = 0.7\%$, with an overall precision loss of only 0.2%. Therefore, DebloaterX is capable of preserving nearly all of the precision of $k$obj.

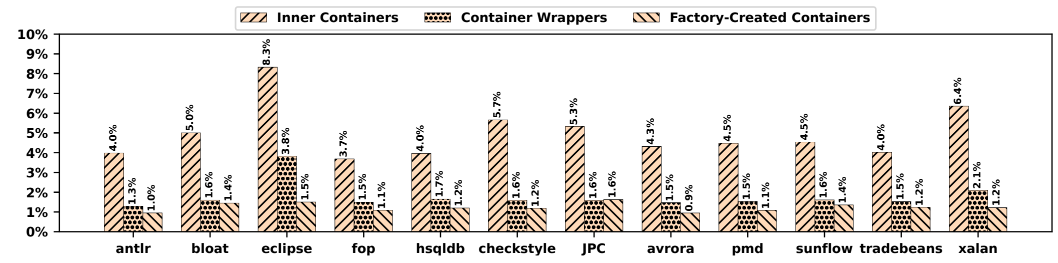
DebloaterX achieves almost the same level of precision as $k$obj by exploiting object sensitivity where it is most effective, namely in scenarios where containers are used according to the three container-usage patterns introduced in Section 2. Moreover, DebloaterX allows $k$obj to avoid context sensitivity for objects that do not have fields of open types, resulting in only negligible precision loss. This is an interesting new discovery that sheds light on the role of context sensitivity in object-sensitive pointer analysis. The rare cases that cause DebloaterX to lose a negligible amount of precision will be discussed by using several examples in RQ4 (Section 5.4).

Table 2. Main analysis results. The results of Spark are also included. **OoM** and **OoT** respectively stand for "Out of Memory" and "Out of Time". For all metrics, smaller is better.

Program		Metrics	Spark	2obj	Z-2obj	S-2obj	C-2obj	X-2obj	3obj	Z-3obj	S-3obj	C-3obj	X-3obj
DaCapo 2006	antlr	Time (s)	6.9	51.0	25.5	10.7	12.8	10.5	2201.3	504.2	11.6	292.6	14.7
		#fail-casts	1127	511	529	597	511	511	451	474	540	451	451
		#call-edges	57472	51319	51446	51433	51319	51319	51292	51419	51394	51292	51295
		#reachables	8194	7806	7836	7834	7806	7806	7805	7835	7834	7805	7805
		#poly-calls	1987	1643	1662	1659	1643	1643	1636	1655	1642	1636	1639
	bloat	Time (s)	8.3	2245.4	1743.2	1612.0	1395.4	26.0	**OoT**	36961.7	28733.0	14547.8	34.4
		#fail-casts	2088	1316	1337	1500	1316	1316	-	1247	1421	1223	1223
		#call-edges	67856	56837	57044	57427	56837	56839	-	56809	57169	56602	56607
		#reachables	9464	9021	9063	9069	9021	9021	-	9047	9053	9005	9005
		#poly-calls	2344	1714	1745	1750	1714	1714	-	1725	1720	1694	1697
	eclipse	Time (s)	31.7	8268.5	2674.7	3097.2	4509.2	1497.4	**OoM**	**OoM**	**OoM**	**OoM**	**OoM**
		#fail-casts	5114	3648	3688	3974	3648	3648	-	-	-	-	-
		#call-edges	183288	162934	163089	163074	162934	162934	-	-	-	-	-
		#reachables	23387	22628	22644	22654	22628	22628	-	-	-	-	-
		#poly-calls	10738	9773	9800	9799	9773	9773	-	-	-	-	-
	fop	Time (s)	5.8	19.7	8.6	4.5	6.4	4.9	1643.0	262.2	5.3	258.4	8.3
		#fail-casts	914	396	416	472	396	396	337	370	414	337	337
		#call-edges	40558	34424	34556	34527	34424	34424	34404	34536	34491	34404	34406
		#reachables	8001	7591	7621	7619	7591	7591	7591	7621	7619	7591	7591
		#poly-calls	1223	842	864	856	842	842	836	858	840	836	838
	hsqldb	Time (s)	5.9	23.1	9.2	4.5	7.4	5.2	2834.0	347.2	5.6	438.6	9.5
		#fail-casts	922	408	428	498	408	408	356	381	449	356	356
		#call-edges	41841	34936	35075	35052	34936	34936	34909	35048	35013	34909	34912
		#reachables	7389	6981	7015	7009	6981	6981	6980	7014	7009	6980	6980
		#poly-calls	1213	859	880	875	859	859	852	873	858	852	855
Non-DaCapo Apps	checkstyle	Time (s)	11.0	9542.1	2759.3	6148.1	6396.0	33.0	**OoM**	**OoT**	**OoM**	**OoT**	47.0
		#fail-casts	1941	1117	1139	1120	1117	1117	-	-	-	-	1005
		#call-edges	80291	67285	67474	67355	67285	67285	-	-	-	-	66543
		#reachables	12773	12315	12350	12341	12315	12315	-	-	-	-	12268
		#poly-calls	2778	2241	2270	2244	2241	2241	-	-	-	-	2197
	JPC	Time (s)	13.7	137.3	42.5	73.0	68.5	61.8	4537.5	368.3	726.7	419.3	78.8
		#fail-casts	2254	1357	1379	1359	1357	1357	1207	1239	1207	1207	1207
		#call-edges	95055	81465	81653	81562	81478	81488	79797	79985	79874	79810	79818
		#reachables	16144	15556	15585	15580	15556	15557	15209	15236	15233	15209	15210
		#poly-calls	4960	4282	4318	4303	4283	4288	4146	4183	4157	4147	4150
DaCapo-9.12	avrora	Time (s)	10.3	336.3	213.2	11.4	18.5	11.5	**OoM**	**OoM**	14.8	2292.9	21.5
		#fail-casts	1215	659	712	723	663	663	-	-	653	584	584
		#call-edges	60549	53799	53871	53810	53799	53799	-	-	53701	53677	53679
		#reachables	12181	11828	11836	11828	11828	11828	-	-	11818	11817	11817
		#poly-calls	1573	1247	1275	1248	1247	1247	-	-	1222	1219	1221
	pmd	Time (s)	11.1	466.9	279.0	193.8	169.7	14.2	**OoM**	**OoM**	167.1	3246.6	25.0
		#fail-casts	1732	1054	1099	1213	1054	1054	-	-	1127	961	963
		#call-edges	60980	53167	53215	53213	53167	53167	-	-	53111	53049	53051
		#reachables	11427	11058	11064	11064	11058	11058	-	-	11060	11053	11053
		#poly-calls	1976	1537	1561	1559	1537	1537	-	-	1543	1519	1521
	sunflow	Time (s)	16.5	922.0	574.2	23.7	28.4	21.7	**OoM**	**OoM**	126.8	2233.2	44.6
		#fail-casts	2269	1384	1452	1553	1388	1388	-	-	1471	1291	1291
		#call-edges	80511	70122	70179	70165	70123	70127	-	-	69829	69787	69793
		#reachables	15733	15289	15295	15291	15289	15290	-	-	15273	15272	15273
		#poly-calls	2832	2360	2390	2364	2360	2360	-	-	2341	2335	2337
	tradebeans	Time (s)	10.5	1066.3	634.1	13.9	22.8	14.9	**OoM**	**OoM**	18.3	6105.6	32.7
		#fail-casts	1262	648	713	726	652	653	-	-	653	564	565
		#call-edges	57400	49476	49645	49579	49476	49476	-	-	49161	49070	49072
		#reachables	10438	10009	10016	10017	10009	10009	-	-	9986	9979	9979
		#poly-calls	1742	1416	1460	1425	1416	1416	-	-	1406	1400	1402
	xalan	Time (s)	14.7	1142.9	675.0	276.7	713.0	33.7	**OoM**	**OoM**	717.7	4579.2	50.8
		#fail-casts	2033	1142	1206	1387	1146	1146	-	-	1315	1065	1065
		#call-edges	80874	71864	71947	71898	71864	71864	-	-	71767	71717	71719
		#reachables	14219	13808	13815	13808	13808	13808	-	-	13801	13800	13800
		#poly-calls	3863	3349	3392	3352	3349	3349	-	-	3335	3330	3332



(a) Percentage of container objects and context-dependent objects



(b) Percentage distributions of context-dependent objects in terms of their container-usage patterns

Fig. 13. Percentages of different categories of objects in a program.

Efficiency. Table 2 reports the analysis times of X-$k$obj and $k$obj. Our results demonstrate that X-$k$obj delivers significant speedups (geometric means) over $k$obj for both $k = 2$ and $k = 3$. Specifically, when $k = 2$, the speedups of X-2obj over 2obj range from 2.2× (for `JPC`) to 289.2× (for `checkstyle`), with an average of 19.3×. When $k = 3$, the speedups on the four benchmarks (`antlr`, `fop`, `hsqldb`, and `JPC`) that can be analyzed scalably by 3obj are impressive, ranging from 57.6× to 298.3×, with an average speedup of 150.2×. For the remaining eight benchmarks, 3obj has failed to analyze them due to either running out of time or memory. In contrast, X-3obj can analyze seven of them within 1–2 minutes. Only `eclipse` cannot be analyzed successfully by X-3obj due to running out of memory, as will be discussed further in RQ4 (Section 5.4). Compared to Spark, the average time used by X-$k$obj on all scalable configurations has only increased to 2.5×.

The impressive acceleration achieved by X-$k$obj over $k$obj is primarily due to DebloaterX's precise selection of a smaller set of objects for context-sensitive analysis performed by X-$k$obj. As shown in Figure 13a, DebloaterX identifies only 26.6% of objects as containers and 7.6% of objects as context-dependent, making it possible to debloat contexts in $k$obj for approximately 92.4% of objects, on average. Furthermore, Figure 13b highlights the distribution of these context-dependent objects according to the three container-usage patterns, where on average, 4.8% are inner containers, 1.7% are container wrappers, and 1.2% are factory-created containers. Note that container wrappers are mutually exclusive with factory-created containers, but have a slight overlap with inner containers (0.2% on average), resulting in the sum of all three categories representing nearly the percentage of context-dependent objects in a program, on average.

We have three additional pieces of evidence to provide support for the impressive acceleration achieved by DebloaterX when comparing X-2obj with 2obj as the normalized baseline, as shown in Figure 14. On average, X-2obj experiences a significant reduction of 90.2% for the number of OAG edges due to context debloating, resulting in substantially fewer contexts being generated (by 61.5%

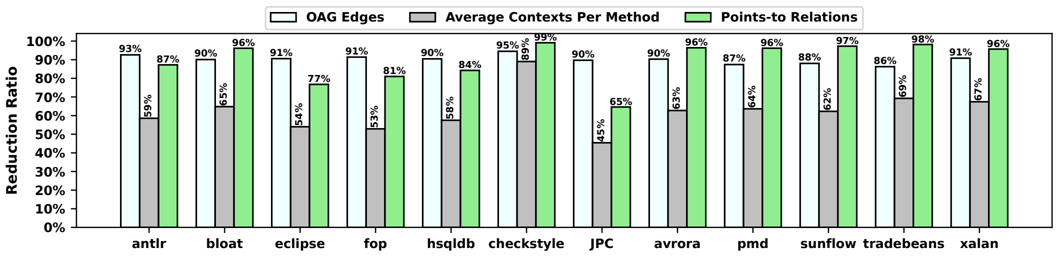


Fig. 14. The reduction ratios of X-2obj relative to 2obj in terms of the number of OAG edges, the average number of calling contexts analyzed per method, and the (context-sensitive) points-to relations.

per method), and consequently, an average reduction of 88.7% for the number of (context-sensitive) points-to relations in a program. We have computed the Spearman rank-order correlation coefficient between the speedups and points-to-relation reduction using `scipy.stats.spearmanr` in Python, which yields a high positive correlation of 0.85 with a p-value of 0.05%.

We have conducted additional investigations into `bloat` in order to understand the exceptional performance of X-2obj on specific benchmarks such as `bloat` and `checkstyle`. Our findings revealed that DebloaterX effectively detected approximately 200 additional context-independent objects allocated within the `EDU.purdue.cs.bloat.tree` package compared to Conch [He et al. 2021a, 2023a]. However, performing context-sensitive analysis on these 200 objects would substantially increase the number of contexts associated with X-2obj from 98,610 to 150,069 (an increase of 52.8%), resulting in prolonged analysis time with limited precision improvement. This phenomenon on `bloat` reveals two facts: (1) the context-dependent objects, which impact the performance of object-sensitive pointer analysis, distribute unevenly within the program; and (2) when analyzed with context-sensitivity, interactions among a limited number of objects can trigger a context-explosion issue, leading to a substantial increase in analysis time.

Robustness. We assess the robustness of DebloaterX mainly by checking if the precision losses of X-$k$obj relative to $k$obj on the three groups of benchmarks from different sources are within an acceptable level (e.g., $\leq 0.5\%$). Note that a significance test in statistics cannot be performed here due to the small sample size. The overall precision losses in "`DaCapo2006`", "`non-DaCapo Apps`", and "`DaCapo-9.12`" are respectively 0.1%, 0.2%, and 0.1%, all of which are at very low levels.

In addition, DebloaterX can significantly boost the performance of $k$obj on all three benchmark groups. For instance, X-2obj achieves average speedups of 8.4× for "`DaCapo2006`", 25.3× for "`Non-DaCapo Apps`", and 39.7× for "`DaCapo-9.12`". The greater speedup observed on "`DaCapo-9.12`" suggests that DebloaterX performs particularly well on larger and more complex benchmarks.

Based on the analysis presented, we can conclude that our approach demonstrates robustness.

5.2 RQ2: Comparing DebloaterX with Two State-of-the-Art Alternatives

We compare DebloaterX with two state-of-the-art pre-analyses, Zipper [Li et al. 2018] and Conch [He et al. 2021a], with respect to improving the precision and efficiency of $k$obj. Figure 15 (Figure 16) presents the average precision loss computed by Equation (2) for the four metrics used (speedups) of X-2obj, C-2obj, Z-2obj and S-2obj for each benchmark, using the data taken from Table 2.

5.2.1 DebloaterX vs. Zipper. When it comes to precision, Zipper causes 2obj to suffer from a small but noticeable precision loss. As shown in Figure 15, the precision loss of Z-2obj ranges from 2.1% (for `eclipse`) to 7.0% (for `tradebeans`) with an average of 4.5% for the 12 programs. However, the precision loss of X-2obj is much lower, with an average of 0.1% across the 12 programs, and the

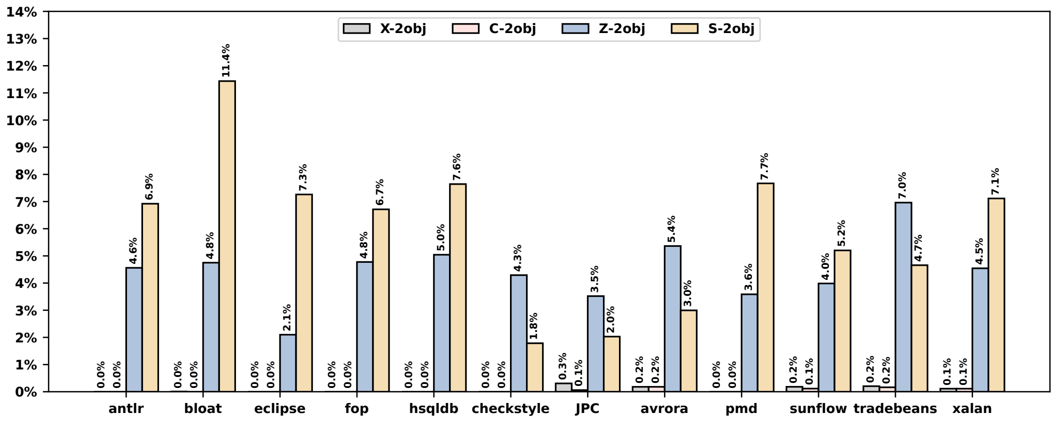


Fig. 15. The average precision loss of X-2obj, C-2obj, Z-2obj and S-2obj on the four selected metrics in Table 2.

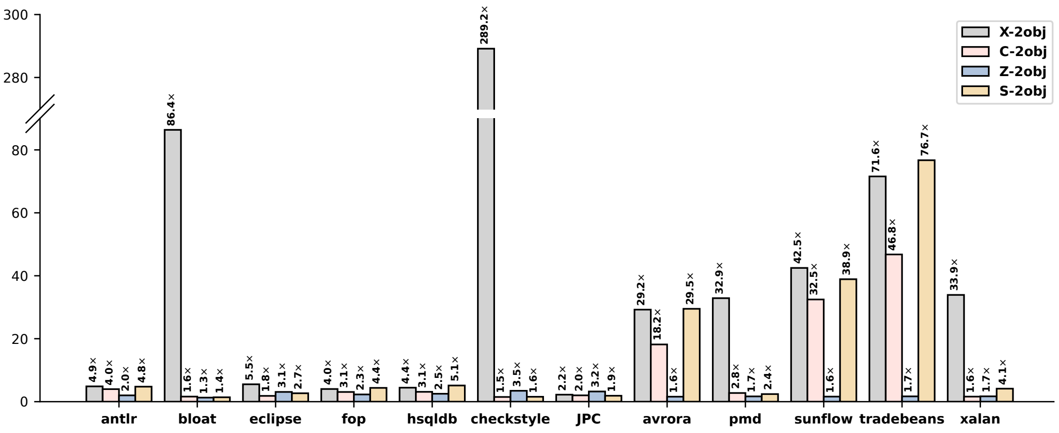


Fig. 16. The speedups of X-2obj, C-2obj, Z-2obj, and S-2obj over 2obj based on the analysis times in Table 2.

largest loss of only 0.3% for JPC. In addition, based on the precision-related data for Z-3obj and X-3obj provided in Table 2, it can be concluded that Zipper is also less precise than DebloaterX.

When it comes to efficiency, X-$k$obj is notably faster than Z-$k$obj. As shown in Figure 16, the speedups of Z-2obj over 2obj range from 1.3× (for `bloat`) to 3.5× (for `checkstyle`), with an average of 2.1× across all programs. However, as discussed in Section 5.1, the average speedup of X-2obj over 2obj is 19.3×, which is substantially larger. When $k = 3$, X-3obj can analyze six more benchmarks scalably than Z-3obj, which are `checkstyle`, `avrora`, `pmd`, `sunflow`, `tradebeans`, and `xalan`. For the four benchmarks that can be analyzed by 3obj as well as X-3obj and Z-3obj, namely `antlr`, `fop`, `hsqldb`, and JPC, the average speedup of X-3obj over 3obj is 150.2×, which significantly exceeds that of Z-3obj (7.2×).

Finally, we can conclude that conducting object-sensitive pointer analysis using DebloaterX results in higher precision and efficiency compared to using Zipper.

Table 3. Times spent by Spark and pre-analyses (DebloaterX, Zipper, and Conch) in seconds.

	antlr	bloat	eclipse	fop	hsqldb	checkstyle	JPC	avrora	pmd	sunflow	tradebeans	xalan
Spark	6.9	8.3	31.7	5.8	5.9	11.0	13.7	10.3	11.1	16.5	10.5	14.7
DebloaterX	13.5	20.2	120.1	10.3	9.4	36.6	41.8	23.4	19.1	39.7	19.0	40.1
Zipper	12.2	13.6	355.5	9.4	8.2	67.3	25.1	13.6	19.1	14.1	7.8	15.2
Conch	8.0	11.5	178.8	4.3	6.4	20.1	18.4	20.3	16.4	32.8	17.6	26.5

5.2.2 DebloaterX vs. Conch. As the first pre-analysis technique for context debloating in $k$obj, Conch can preserve almost all of its precision [He et al. 2021a, 2023a] and in this paper. Our comparison of X-2obj and C-2obj in terms of average precision loss (0.1%), as shown in Figure 15, indicates that they are comparable. However, X-2obj suffers slightly, but negligibly, more precision loss on some benchmarks (e.g., `JPC` and `sunflow`) due to some precision-critical objects being identified as context-independent by DebloaterX but context-dependent by Conch. Conversely, some objects are identified as context-dependent by DebloaterX but incorrectly as context-independent by Conch, which happens to not bring noticeable precision benefits to DebloaterX. The relative precision loss of X-3obj compared to C-3obj follows the same trend, as seen in Table 2.

In terms of efficiency, DebloaterX outperforms Conch in accelerating $k$obj, providing better speedups and scalability. As shown in Figure 16, the average speedups of C-2obj over 2obj range from 1.5× (for `checkstyle`) to 46.8× (for `tradebeans`) with an average of 4.3× across the 12 programs. In contrast, X-2obj outperforms C-2obj on all 12 benchmarks, resulting in a larger average speedup over 2obj (19.3×). For $k = 3$, both C-3obj and X-3obj are fast and enable significantly more benchmarks to be analyzed scalably than Z-3obj and 3obj. Specifically, X-3obj can analyze 11 out of 12 programs with the exception of `eclipse`, while C-3obj has only been successful in analyzing 10 programs, with the additional failure being on `checkstyle`. However, among the 11 programs analyzed by both, X-3obj typically completes a benchmark within 1-2 minutes, making it significantly faster than C-3obj. Direct comparison between X-3obj and C-3obj shows that the speedups of X-3obj over C-3obj range from 5.3× (for `JPC`) to 422.9× (for `bloat`), with an average of 61.3×.

The impressive performance of X-3obj over C-3obj is due to three key factors: (1) DebloaterX employs a 1-limited field-sensitive approach to identify container objects, which leads to higher precision than Conch (which is field-insensitive), (2) our open type filtering mechanism enables DebloaterX to remove a set of container objects that are theoretically context-dependent but not practically context-sensitive, without sacrificing precision, and (3) the use of three container-usage patterns significantly reduces the number of false-positive context-dependent objects identified by Conch. All three factors contribute to the improved performance of DebloaterX, which selects an average of only 7.6% of objects as context-dependent, much lower than Conch's average of 14.9%.

5.2.3 Pre-Analysis Overheads. DebloaterX is an efficient pre-analysis tool, with similar overheads to Zipper and Conch. All three tools are designed to be multithreaded (using 16 threads in our experiments), with their times shown in Table 3 (including the analysis time of Spark for comparison purposes). On average (geomeans), DebloaterX takes 25.2 seconds, while Zipper and Conch take 19.7 and 17.5 seconds, respectively. Despite supporting some field sensitivity, DebloaterX is only slightly (6-7 seconds on average) slower than both Zipper and Conch, and the additional overhead is negligible considering the overall reduction in analysis time achieved by $k$obj.

5.3 RQ3: Comparing DebloaterX with a Simplistic Pre-Analysis

DebloaterX is specifically designed to significantly enhance the performance of $k$obj while preserving nearly all of its precision. To further highlight its necessity, we compare DebloaterX with a simplistic pre-analysis approach, denoted SimCon, based on `java.lang.Collection`, as defined at the beginning of Section 5. SimCon identifies context-dependent objects by checking if they belong to container types. In this context, a container type refers to a Java class that either contains a `java.lang.Object` field, has a container type field (recursively), implements the `java.util.Collection` interface, or is nested within a class implementing `java.util.Collection`.

SimCon has caused $k$obj to suffer from a significant loss of precision compared to DebloaterX, as shown in Table 2. Further analysis (with $k = 2$), as depicted in Figure 15, reveals that the precision loss of S-2obj ranges from 1.8% (for `checkstyle`) to 11.4% (for `bloat`), with an average loss of 6.0% across the 12 programs. This average loss is substantially higher compared to that observed with DebloaterX. As discussed in Section 5.2, the precision loss of X-2obj is considerably lower, with an average loss of 0.1% across these 12 programs. The largest precision loss observed is a mere 0.3% for `JPC`. A similar trend can be observed when $k = 3$.

As revealed in our experimental results, X-$k$obj is not only substantially more precise than S-$k$obj but also substantially more efficient overall. The average speedup achieved by X-$k$obj is significantly higher than that of S-$k$obj across the 12 programs, despite the fact that S-$k$obj may occasionally run faster than X-$k$obj for certain programs. As illustrated in Figure 16 (with $k = 2$), S-2obj runs marginally faster than X-2obj in four benchmarks: `fop`, `hsqldb`, `avrora`, and `tradebeans` (by 1.1× on average). However, for the remaining eight benchmarks, X-2obj is faster than S-2obj (by 6.6× on average). Overall, the speedups observed for S-2obj over the baseline 2obj range from 1.4× (for `bloat`) to 76.7× (for `tradebeans`), with an average speedup of 5.6×. In contrast, the average speedup achieved by X-2obj over 2obj is substantially higher at 19.3× compared to S-2obj. When $k = 3$, X-3obj can scalably analyze all 12 programs, except for `eclipse`. In comparison, S-3obj can handle one fewer program, namely `checkstyle`, in a scalable manner. Among the 10 programs that can be analyzed by both approaches, the average speedup attained by X-3obj over S-3obj is 3.5×. In particular, among the four benchmarks that can be scalably analyzed by 3obj, the average speedup achieved by S-3obj over the baseline 3obj is 116.8×. This speedup is noticeably lower compared to the speedup achieved by X-3obj over the same baseline 3obj, which amounts to 150.2×.

Therefore, we conclude that simple-minded pre-analysis approaches like SimCon are inadequate substitutes for our advanced DebloaterX approach in achieving the design objective of developing an effective context-debloating technique that enhances the performance of $k$obj while maintaining nearly all of its precision. Furthermore, the impact of SimCon on the runtime performance of $k$obj compared to DebloaterX is highly unpredictable, as SimCon relies on simplistic heuristics to randomly distinguish between context-dependent and context-independent objects.

5.4 RQ4: The Precision Loss and Scalability Issues of X-$k$obj

We have demonstrated that DebloaterX can significantly improve the performance of X-$k$obj over $k$obj with only a negligible loss of precision (often less than 0.2%). While rare cases do exist where X-$k$obj may lose precision, we will discuss those first before examining why X-3obj is still unable to analyze `eclipse` within 12 hours, which provides some insights for future research.

X-$k$obj sometimes loses precision due to context-dependent objects that are missed by DebloaterX. Four rare but representative cases are presented in Figure 17. In Figure 17a, a `TimSort` object, `T`, created in line 8 is identified as a container object but not as context-dependent because its usage pattern does not fall into any of the three container-usage categories considered. However, there are incoming and outgoing value flows on the field `a` of `T`, causing a slight loss of precision in both DebloaterX and Conch on the `DaCapo-9.12` benchmarks.

In Figure 17b, a `Pattern` object named `P`, created in line 4, has not been identified successfully as a factory-created container because DebloaterX cannot recognize it as such, given that all its fields are of concrete types. However, the type of the field `root` of `P`, `Node`, has over 50 subtypes

```
1  class TimSort { // java.util;
2    Object[] a, tmp;
3    TimSort(Object[] p1) {
4      this.a = p1;
5    }
6
7    static void sort(Object[] q1) {
8      TimSort timsort = new TimSort(q1); // T
9      timsort.pushRun();
10     timsort.mergeCollapse();
11   }
12 }
```

(a) An uncovered container usage pattern

```
1 class Pattern { // java.util.regex;
2   Node root; // hasInFlow and hasOutFlow
3   static Pattern compile(String p) {
4     return new Pattern(p, 0); // P
5 }}
```

(b) An omitted factory-created container

```
1 abstract class Provider {// java.security;
2   Map legacyMap;
3   void parseLegacyPut(String p) {
4     Service s = new Service(this); // S
5     this.legacyMap.put (...,  s)
6 }}
```

(c) An omitted inner container.

```
1 class A {
2   Object id(Object p) { return p; }
3   static Object wid(Object q) {
4     return new A().id(q); // A
5 }}
```

```
1 class B {
2   void foo(Object o1) { v1 = wid(o1); }
3   void bar(Object o2) { v2 = wid(o2); }}
4 new B().foo(new Object()); // B1, O1
5 new B().bar(new Object()); // B2, O2
```

(d) A handcrafted case

Fig. 17. Rare cases where DebloaterX losses precision.

defined in `java.util.regex.Pattern`, making it as abstract as an open type (defined in Figure 6), causing precision loss. We plan to address such cases in future work by developing heuristics.

In Figure 17c, DebloaterX fails to recognize object S as an inner container. Created in line 4, S is stored into `this.legacyMap.*` via a virtual call to `put()` in line 5. DebloaterX models parameter passing at virtual calls imprecisely using cstore and cload edges in [X-Virtual]. However. the NFA used for identifying inner containers (in Figure 10) excludes cstore and cload edges (as well as their inverses), preventing S from reaching the NFA's final state. This design choice represents a trade-off since including cstore and $\overline{\text{cload}}$ (just like for store and $\overline{\text{load}}$) in the NFA would lead to significantly reduced performance improvements by falsely identifying more context-dependent objects.

We have shown three cases where DebloaterX loses precision in real-world applications. Let us examine a handcrafted example in Figure 17d where precision loss can theoretically occur under object sensitivity. However, this theoretical case has not been observed practically by us. In the example, O1 (O2) is written into `A.p` under context `[B1]` (`[B2]`) and returned to and stored in `v1` (`v2`). Analyzing `A` context-insensitively would cause O1 (O2) to be pointed to spuriously by `v2` (`v1`). However, DebloaterX selects `A` as context-independent, causing $k$obj to lose precision.

Finally, X-3obj fails to analyze `eclipse` within a 12-hour budget, despite DebloaterX selecting 86.7% of objects in `eclipse` as context-independent. The remaining 2929 objects whose contexts are not debloated during the analysis have caused X-3obj to run out of memory.

6 RELATED WORK

In this paper, we have introduced DebloaterX, a novel context-debloating technique designed to accelerate object-sensitive pointer analysis. While the idea of context-debloating was proposed in previous work [He et al. 2021a, 2023a], our container-usage-patterns-based approach is original and significantly more efficient than Conch, with almost the same precision.

Several approaches have been proposed to enhance the efficiency of context-sensitive pointer analysis by selectively applying context sensitivity to a subset of variables and objects [Hassanshahi et al. 2017; He et al. 2021b, 2023b; Jeong et al. 2017; Li et al. 2018, 2020; Lu et al. 2021a,b; Lu and Xue

2019; Smaragdakis et al. 2014]. However, their goal is to reduce the number of analyzed variables and objects rather than the number of generated contexts, which limits their performance improvements. While Conch [He et al. 2021a, 2023a] was the first to reduce the number of generated contexts, DebloaterX is the first to leverage container-usage patterns. This approach results in significantly improved efficiency while maintaining similar precision compared to Conch.

There are several different kinds of context-sensitivity being proposed in the literature, including call-site sensitivity [Shivers 1991], object sensitivity [Milanova et al. 2002, 2005], type sensitivity [Smaragdakis et al. 2011], hybrid sensitivity [Kastrinis and Smaragdakis 2013], generic sensitivity [Li et al. 2022], context transformation [Thiessen and Lhoták 2017], and others [Tan et al. 2017; Thakur and Nandivada 2020]. These approaches study which kinds of context elements should be used to achieve a balance between precision and efficiency in pointer analysis. Among these approaches, object-sensitivity [Milanova et al. 2002, 2005] has been demonstrated to provide the most effective context abstraction for object-oriented languages like Java, particularly when k-limiting is enforced. A recent advancement in optimizing object-sensitive pointer analysis through the development of object-sensitive library summaries is detailed in [Lu et al. 2023].

Context tunneling [Jeon et al. 2018] and Bean [Tan et al. 2016] are techniques that seek to enhance the precision of pointer analysis by choosing which context elements to preserve while constructing new contexts within a specified context-length limit under k-limiting. This line of research differs from the work presented in this paper, which focuses on context-debloating techniques.

Tan et al. [2021] propose a `Unity-Relay` framework, which aims to retain the precision preserved by at least one of a set of selective context-sensitivity techniques in order to achieve the best precision among all provided techniques. However, this approach may be less effective when used in conjunction with approaches that are already precision-preserving [Lu and Xue 2019] or nearly precision-preserving [He et al. 2021b,a; Li et al. 2018], as well as with DebloaterX.

Pointer analysis has also been extensively studied from the perspective of CFL-reachability, both for improving the efficiency of alias analysis [Zhang et al. 2013, 2014] and for supporting demand-driven pointer analysis [Sridharan and Bodík 2006; Sridharan et al. 2005; Yan et al. 2011] selective context-sensitivity [He et al. 2021b; Lu et al. 2021a,b; Lu and Xue 2019]. In addition, incremental techniques have also been investigated recently [Liu and Huang 2022; Liu et al. 2019].

Finally, pointer analysis has also been studied from the perspective of understanding their time and space complexities [Mathiasen and Pavlogiannis 2021; Sridharan and Fink 2009]. In contrast, this paper provides an approach aiming to make pointer analysis practically usable.

7 CONCLUSION AND FUTURE WORK

This paper introduces DebloaterX, a novel context-debloating approach that uses three container-usage patterns to improve the efficiency of object-sensitive pointer analysis. We have developed precise and efficient rules and algorithms to identify these patterns. Our evaluation shows that DebloaterX can accelerate pointer analysis by one to two orders of magnitude with minimal loss of precision, outperforming existing state-of-the-art pre-analysis techniques.

Improving the efficiency of context-sensitive pointer analysis for object-oriented programming languages by applying context-debloating is a promising approach. We have identified two areas for future work: (1) extending DebloaterX to make it pluggable for object-usage patterns that may not have been captured in this paper, and (2) developing better context-debloating techniques to enable scalable analysis of large and complex programs, such as `eclipse`, in practice.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable comments. This research is supported by ARC Grant DP210102409.

REFERENCES

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167515.1167488

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*. IEEE, Honolulu, HI, USA, 241–250. https://doi.org/10.1145/1985793.1985827

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/1639949.1640108

Yuandao Cai, Chengfeng Ye, Qingkai Shi, and Charles Zhang. 2022. Peahen: Fast and Precise Static Deadlock Detection via Context Reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 784–796. https://doi.org/10.1145/3540250.3549110

Isabel Garcia-Contreras, Arie Gurfinkel, and Jorge A Navas. 2022. Efficient Modular SMT-Based Model Checking of Pointer Programs. In *Static Analysis: 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, Proceedings*. Springer Nature Switzerland, Cham, 227–246. https://doi.org/10.1007/978-3-031-22308-2_11

Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An Efficient Tunable Selective Points-to Analysis for Large Codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3088515.3088519

Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2021b. Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:31. https://doi.org/10.4230/LIPIcs.ECOOP.2021.16

Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2023b. Selecting Context-Sensitivity Modularly for Accelerating Object-Sensitive Pointer Analysis. *IEEE Transactions on Software Engineering* 49, 2 (2023), 719–742. https://doi.org/10.1109/TSE.2022.3162236

Dongjie He, Jingbo Lu, and Jingling Xue. 2021a. Context Debloating for Object-Sensitive Pointer Analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, New York, NY, USA, 79–91. https://doi.org/10.1109/ASE51524.2021.9678880

Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A New Framework for Supporting Fine-Grained Context-Sensitivity in Java Pointer Analysis. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.30

Dongjie He, Jingbo Lu, and Jingling Xue. 2023a. IFDS-Based Context Debloating for Object-Sensitive Pointer Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4 (jan 2023), 1–45. https://doi.org/10.1145/3579641

Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. https://doi.org/10.1145/3276510

Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning Graph-Based Heuristics for Pointer Analysis without Handcrafting Application-Specific Features. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 179 (nov 2020), 30 pages. https://doi.org/10.1145/3428247

Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: Call-Site Sensitivity Can Be Superior to Object Sensitivity Even for Object-Oriented Programs. *Proc. ACM Program. Lang.* 6, POPL, Article 58 (jan 2022), 29 pages. https://doi.org/10.1145/3498720

Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 100. https://doi.org/10.1145/3133924

George Kastrinis and Yannis Smaragdakis. 2013. Hybrid Context-Sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 423–434. https://doi.org/10.1145/2499370.2462191

Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 153–169. https://doi.org/10.1007/3-540-36579-6_12

Haofeng Li, Jie Lu, Haining Meng, Liqing Cao, Yongheng Huang, Lian Li, and Lin Gao. 2022. Generic sensitivity: customizing context-sensitive pointer analysis for generics. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 1110–1121. https://doi.org/10.1145/3540250.3549122

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29. https://doi.org/10.1145/3276511

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Transactions on Programming Languages and Systems* 42, TOPLAS (2020), 1–40. https://doi.org/10.1145/3381915

Bozhen Liu and Jeff Huang. 2022. SHARP: Fast Incremental Context-Sensitive Pointer Analysis for Java. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 88 (apr 2022), 28 pages. https://doi.org/10.1145/3527332

Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking incremental and parallel pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2019), 1–31. https://doi.org/10.1145/3293606

Jingbo Lu, Dongjie He, Wei Li, Yaoqing Gao, and Jingling Xue. 2023. Automatic Generation and Reuse of Precise Library Summaries for Object-Sensitive Pointer Analysis. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Jingbo Lu, Dongjie He, and Jingling Xue. 2021a. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–46. https://doi.org/10.1145/3450492

Jingbo Lu, Dongjie He, and Jingling Xue. 2021b. Selective Context-Sensitivity for k-CFA with CFL-Reachability. In *Static Analysis: 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17–19, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 261–285. https://doi.org/10.1007/978-3-030-88806-0_13

Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29. https://doi.org/10.1145/3360574

Anders Alnor Mathiasen and Andreas Pavlogiannis. 2021. The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 34 (jan 2021), 29 pages. https://doi.org/10.1145/3434315

Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/566172.566174

Ana Milanova, Atanas Rountev, and Barbara G Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology* 14, 1 (2005), 1–41. https://doi.org/10.1145/1044834.1044835

Mehryar Mohri and Mark-Jan Nederhof. 2001. Regular Approximation of Context-Free Grammars through Transformation. In *Robustness in Language and Speech Technology*, Jean-Claude Junqua and Gertjan van Noord (Eds.). Springer Netherlands, Dordrecht, 153–163. https://doi.org/10.1007/978-94-015-9719-7_6

Ankush Phulia, Vaibhav Bhagee, and Sorav Bansal. 2020. OOElala: Order-of-Evaluation Based Alias Analysis for Compiler Optimization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 839–853. https://doi.org/10.1145/3385412.3385962

Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 162–186. https://doi.org/10.1145/345099.345137

Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages or taming lambda.* Ph. D. Dissertation. Carnegie Mellon University. CMU-CS-91-145.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/1925844.1926390

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

Manu Sridharan and Rastislav Bodík. 2006. Refinement-Based Context-Sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

Manu Sridharan and Stephen J Fink. 2009. The complexity of Andersen's analysis in practice. In *Static Analysis: 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings 16*. Springer Berlin Heidelberg, Berlin, Heidelberg, 205–221. https://doi.org/10.1007/978-3-642-03237-0_15

Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/1250734.1250748

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-Driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 59–76. https://doi.org/10.1145/

1094811.1094817

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. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27. https://doi.org/10.1145/3485524

Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*. Springer Berlin Heidelberg, Berlin, Heidelberg, 489–510. https://doi.org/10.1007/978-3-662-53413-7_24

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 278–291. https://doi.org/10.1145/3140587.3062360

Manas Thakur and V. Krishna Nandivada. 2020. Mix Your Contexts Well: Opportunities Unleashed by Recent Advances in Scaling Context-Sensitivity. In *Proceedings of the 29th International Conference on Compiler Construction*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/3377555.3377902

Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 263–277. https://doi.org/10.1145/3140587.3062359

David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Past-Sensitive Pointer Analysis for Symbolic Execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/3368089.3409698

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818

WALA. 2023. WALA: T.J. Watson Libraries for Analysis. Retrieved April 8, 2023 from http://wala.sourceforge.net/

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, USA, 155–165. https://doi.org/10.1145/2001420.2001440

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. 2013. Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 435–446. https://doi.org/10.1145/2491956.2462159

Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. 2014. Efficient Subcubic Alias Analysis for C. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 829–845. https://doi.org/10.1145/2660193.2660213

Received 2023-04-14; accepted 2023-08-27