IFDS-based Context Debloating for Object-Sensitive Pointer Analysis

DONGJIE HE, JINGBO LU, and JINGLING XUE, UNSW Sydney, Australia

Object-sensitive pointer analysis, which separates the calling contexts of a method by its receiver objects, is known to achieve highly useful precision for object-oriented languages such as Java. Despite recent advances, all object-sensitive pointer analysis algorithms still suffer from the scalability problem due to the combinatorial explosion of contexts in large programs. In this article, we introduce a new approach, CONCH, that can be applied to debloat contexts for *all* object-sensitive pointer analysis algorithms, thereby improving significantly their efficiency while incurring a negligible loss of precision. Our key insight is to approximate a recently proposed set of two necessary conditions for an object in a program to be context-sensitive, i.e., context-dependent (whose precise verification is undecidable) with a set of three linearly verifiable conditions in terms of the number of edges in the pointer assignment graph (PAG) representation of the program. These three linearly verifiable conditions, which turn out to be almost always necessary in practice, are synthesized from three key observations regarding context-dependability for the objects created and used in real-world object-oriented programs. To develop a practical implementation for CONCH, we introduce an IFDS-based algorithm for reasoning about object reachability in the PAG of a program, which runs linearly in terms of the number of edges in the PAG. By debloating contexts for three representative object-sensitive pointer analysis algorithms, which are applied to a set of representative Java programs, CONCH can speed up these three baseline algorithms substantially at only a negligible loss of precision (less than 0.1%) with respect to several commonly used precision metrics. In addition, CONCH also improves their scalability by enabling them to analyze substantially more programs to completion than before (under a time budget of 12 hours). CONCH has been open-sourced (http://www.cse.unsw.edu.au/~corg/tools/conch), opening up new opportunities for other researchers and practitioners to further improve this research. To demonstrate this, we introduce one extension of CONCH to accelerate further the three baselines without losing any precision, providing further insights on extending CONCH to make precision-efficiency tradeoffs in future research.

101

CCS Concepts: • **Theory of Computation** → Program Analysis;

Additional Key Words and Phrases: Pointer analysis, object sensitivity, context debloating, IFDS

ACM Reference format:
Dongjie He, Jingbo Lu, and Jingling Xue. 2023. IFDS-based Context Debloating for Object-Sensitive Pointer Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 101 (May 2023), 44 pages.
https://doi.org/10.1145/3579641

Authors' address: D. He, J. Lu, and J. Xue, University of New South Wales, High Street, Kensington, New South Wales (NSW), 2052, Australia; emails: {dongjieh, jlu, jingling}@cse.unsw.edu.au.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1049-331X/2023/05-ART101 $15.00
https://doi.org/10.1145/3579641

1 INTRODUCTION

A wide range of software engineering tasks such as call graph construction [21], program slicing [47, 57], program understanding [27], symbolic execution [54], fuzz testing [56], bug detection [2, 12, 28, 36, 60], and security analysis [19, 29] often require precise points-to/alias information about the program under consideration. The quality of a pointer analysis directly determines the effectiveness and usefulness of the tools developed for accomplishing these tasks.

For object-oriented languages such as Java, object-sensitive pointer analysis, which distinguishes the (calling) contexts of a method by its receiver objects, is regarded as providing highly useful precision [16, 32, 44, 52, 53] and thus widely adopted in a number of pointer analysis frameworks for Java, such as Soot [55], Doop [43], Wala [7], and Qilin [15]. Under k-object-sensitivity (i.e., a k-object-sensitive analysis with a $(k-1)$-context-sensitive heap) [34, 35], denoted $k$obj, a context used for analyzing a method m and all variables declared therein is represented by a sequence of k context elements (under k limiting), $[o_1, \ldots, o_k]$, where o_1 is the receiver object of m and o_i is the receiver object of a method in which o_{i-1} is allocated [44]. Therefore, o_i is known as an *allocator* of o_{i-1}. For a heap object (created at an allocation site) in method m modeled with heap cloning in [37], its context (often referred to as its heap context [20, 44]) is represented by $[o_1, \ldots, o_{k-1}]$ (under $k-1$ limiting). Currently, $k$obj does not scale well for reasonably large object-oriented programs when $k \geqslant 3$ and is often time-consuming when it is scalable [16, 44, 52, 53]. As k increases, the number of contexts that are analyzed for a method often blows up exponentially without improving precision much. To alleviate this issue, several recent research efforts [11, 18, 24, 32] focus on applying selective context-sensitivity to a program by first conducting a pre-analysis on the program and then instructing $k$obj to apply context-sensitivity only to some selected methods in the program. A number of attempts have recently been made, including client-specific machine learning techniques [18] (guided by improving the precision of a specific client, e.g., may-fail-casting) and general-purpose techniques, such as user-supplied hints [11], pattern matching [24], and **Context-Free Language** (**CFL**) reachability [13, 30, 32]. Despite some performance improvements obtained (at either no or a noticeable loss of precision), these existing selective context-sensitive pointer analysis algorithms still suffer from an unreasonable explosion of contexts.

In this article, we introduce a new approach, Conch, for debloating contexts for *all* object-sensitive pointer analysis algorithms, including $k$obj and its various incarnations for performing selective context-sensitivity, by boosting their performance significantly while incurring only a negligible loss in precision. In real-world object-oriented programs, we observe that a large number of objects that are allocated in a method are often used independently of its calling contexts. These objects are either used locally such as the ones created in $\texttt{foo}_{i,j}$() and $\texttt{bar}_{i,j}$() in Figure 4 (introduced shortly) or designed to encapsulate only primitive data such as `java.lang.Integer`. Distinguishing these objects context-sensitively, as often done in the current object-sensitive pointer analysis algorithms, serves to increase only the number of calling contexts analyzed for the methods invoked on these objects (as their receivers) without delivering any precision improvement. Therefore, our key insight in developing Conch is to approximate a recently proposed set of two necessary conditions for an object in a program to be context-sensitive, i.e., *context-dependent* [30, 32] (whose precise verification is undecidable [39]) with a set of three linearly verifiable necessary conditions (in terms of the number of edges in the **Pointer Assignment Graph** (**PAG**) of the program), based on three key observations regarding context-dependability for the objects created and used in real-world object-oriented programs. To develop a practical implementation for Conch, we introduce a lightweight IFDS-based algorithm [40] for verifying these three conditions, which govern essentially object reachability in the program. To the best of our knowledge, this is the first IFDS-based algorithm that operates on the PAG instead of the CFG of

a program, and in addition, this algorithm solves the context-debloating problem elegantly and efficiently as it runs linearly in terms of the number of PAG edges in the program. By instructing any given object-sensitive pointer analysis algorithm to analyze all context-independent objects context-insensitively if it is not designed to do so, Conch can enable it to limit effectively the explosive growth of the number of contexts, thereby achieving substantially improved efficiency and scalability at a negligible loss of precision.

We have implemented Conch in Soot [55], a program analysis and optimization framework for Java. We have evaluated Conch by applying it to boost the performance of $k$obj [35], Eagle [32] (a representative precision-preserving selective context-sensitive pointer analysis), and Zipper [24] (a representative non-precision-preserving selective context-sensitive pointer analysis) using a set of popular Java benchmarks and applications. Conch can speed up all three baseline algorithms substantially at no or little loss of precision (less than 0.1%). In addition, Conch can also improve their scalability by enabling them to analyze substantially more programs under more configurations to completion than before (under a time budget of 12 hours).

In summary, this article makes the following contributions:

— We present context debloating, a new approach for accelerating all object-sensitive pointer analysis algorithms while incurring only a negligible loss of precision.
— We give a set of three conditions (which are usually necessary in real code) for determining an object's context-dependability and propose a lightweight IFDS-based algorithm for verifying these conditions (linearly in terms of the number of PAG edges in a program).
— We have extensively evaluated the effectiveness of Conch (using a number of popular metrics) and demonstrated its practical significance for real-world programs.
— We have implemented Conch in the Soot framework [55] and open-sourced it at http://www.cse.unsw.edu.au/~corg/tools/conch. We hope this will open up new opportunities for other researchers and practitioners to further improve this research. To demonstrate this, we introduce one extension of Conch to boost the performance of object-sensitive pointer analysis algorithms further without losing any precision, providing further insights on extending Conch to make various precision-efficiency tradeoffs in future research.

The rest of this article is organized as follows. Section 2 motivates the need for context debloating. Section 3 gives a version of $k$obj that supports context debloating. Section 4 presents our Conch approach. In Section 5, we evaluate the effectiveness of Conch achieved by context debloating. Section 6 discusses the related work. Finally, Section 7 concludes the article.

2 MOTIVATION

We first review the IFDS framework (Section 2.1). We then discuss the context explosion problem in existing object-sensitive pointer analysis algorithms (Section 2.2). Finally, we motivate our context-debloating approach, by describing its basic idea, examining the main challenges faced in realizing it efficiently and effectively, and discussing our solution for addressing these challenges (Section 2.3).

2.1 The IFDS Framework

The IFDS framework introduced by Reps et al. [40] aims to solve a special kind of data-flow problem, called the *inter-procedural, finite, distributive subset problem*, in a flow-sensitive, fully context-sensitive manner, by operating on a supergraph representation of a given program.

Definition 1 (Supergraph). The *supergraph* $G^* = (N^*, E^*)$ of a program consists of a set of control-flow graphs (CFGs), $G_1, G_2, \ldots$ (one for each method and one of which, G_{main}, represents the main method of the program), which are connected by inter-procedural edges:

— N^* is the set of nodes representing program points. For a method m, its CFG G_m has a unique *start-node* $s_m \in N^*$ (*exit-node* $e_m \in N^*$). A callsite is represented by a *call-node* $c \in N^*$ and a *return-node* $r \in N^*$. The other nodes (i.e., *normal nodes*) represent the statements as usual.
— E^* is the set of control-flow edges classified into four kinds: *call edges* (connecting a call-node to a start-node), *return edges* (connecting an exit-node to a return-node), *call-to-return edges* (connecting a call-node to a return-node), and *normal edges* (connecting normal nodes).

An IFDS problem that operates on the supergraph of a program can be formally defined.

Definition 2 (IFDS Problem). An *IFDS problem IP* for a program is a quintuple $IP = (G^*, D, F, M, \sqcap)$, where G^* is the supergraph of the program, D is a finite set of data-flow facts, $F \subseteq 2^D \rightarrow 2^D$ is a set of distributive functions, $M : E^* \mapsto F$ is a map from E^* to data-flow functions, and the meet operator $\sqcap$ is either union or intersection (depending on the problem modeled).

Reps et al. [40] propose an efficient tabulation algorithm to solve an IFDS problem precisely by transforming it into a special kind of graph-reachability problem on an exploded supergraph.

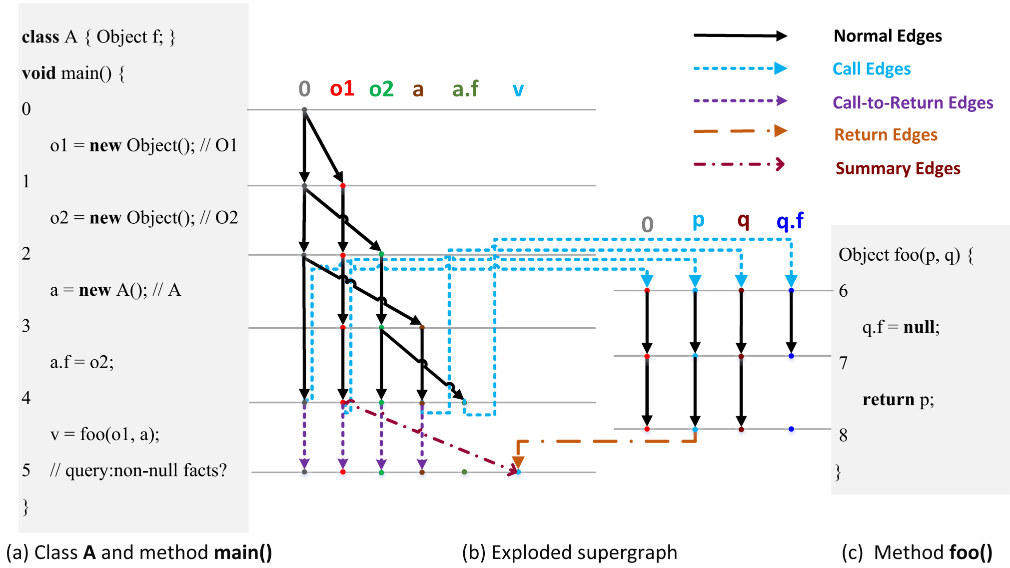
Definition 3 (Exploded Supergraph). Let $IP = (G^*, D, F, M, \sqcap)$ be an IFDS problem. The corresponding *exploded supergraph* $G^\#_{IP} = (N^\#, E^\#)$ is defined as follows:

— $N^\# = N^* \times (D \cup \mathbf{0})$, where $\mathbf{0}$ signifies an empty set of data-flow facts (allowing new data-flow facts to be generated at a program point).
— $E^\# = \{\langle n_1, d_1\rangle \rightarrow \langle n_2, d_2\rangle \mid (n_1, n_2) \in E^* \wedge d_2 \in f(d_1)\}$, where $f = M(n_1, n_2)$ is the flow function associated with the edge $(n_1, n_2) \in E^*$.

For a program, an exploded supergraph is, therefore, a graph extended from its supergraph with the flow functions being explicitly represented. As a result, the problem of checking whether a data-flow fact $d \in D$ is available at a given program point $n \in N^*$ in a method m is equivalent to one of checking whether $\langle n, d\rangle \in N^\#$ is reachable from $\langle s_{main}, \mathbf{0}\rangle$. This is represented by a so-called *path edge* $\langle s_m, d'\rangle \rightarrow \langle n, d\rangle$, with the understanding that a realizable path from $\langle s_{main}, \mathbf{0}\rangle$ to $\langle s_m, d'\rangle$ always exists. A program path is *realizable* if its method returns always match with their corresponding calls. Due to the distributivity of flow functions, the IFDS algorithm can summarize and reuse the effects of methods, thereby boosting significantly its efficiency. The time complexity of the IFDS algorithm is $O(|E^*| \cdot |D|^3)$. In practice, the IFDS algorithm usually makes use of a pre-built call graph, with the aliasing information discovered either beforehand or on the fly [2].

Figure 1 illustrates how the IFDS algorithm performs a non-null value analysis for a program by answering the query regarding the non-null (data-flow) facts available at program point 5 (i.e., at the end of the program). In `main()`, the facts `o1`, `o2`, and `a` start being non-null after program points 1, 2, and 3, respectively, and will be available at program point 5. The fact `a.f` becomes non-null after program point 4 due to `a.f = o2`. Subsequently, `a` is passed into `foo()` (via its parameter `q`), making `q.f` non-null immediately. However, due to `q.f = null`, `q.f` becomes null after program point 7. As a result, `a.f` is no longer available at program point 5. In contrast, `o1` is passed into `foo()` (via its parameter `p`) and then returned to `v`. Thus, `v` is available at program point 5.

In this example, answering the given query at program point 5 amounts to checking whether the data flow facts in {`o1`, `o2`, `a`, `a.f`, `v`} are reachable from $\langle s_{main}, \mathbf{0}\rangle$ in its exploded supergraph. We can see clearly that all these facts except `a.f` are available at program point 5, since, for each $d \in \{$`o1`, `o2`, `a`, `v`$\}$, the path edge $\langle s_{main}, \mathbf{0}\rangle \rightarrow \langle 5, d\rangle$ exists. It is important to point out that one summary edge, $\langle 4, \texttt{o1}\rangle \rightarrow \langle 5, v\rangle$, has been added to the callsite `v = foo(o1, a)` (as depicted). If there were another callsite, say, `v′ = foo(o1′, a′)` made between program points n and $n+1$ in the program (if any), a summary edge, $\langle n, \texttt{o1}'\rangle \rightarrow \langle n+1, v'\rangle$ would also be added so that the same reachability information for a method is discovered only once and reused across its callsites.



(a) Class **A** and method **main()** (b) Exploded supergraph (c) Method **foo()**

Fig. 1. An illustration of how the IFDS algorithm performs a non-null value analysis for a program given in (a) and (c) on its exploded supergraph (with only its visited edges generated (on the fly)).

2.2 Existing Object-Sensitive Pointer Analysis Algorithms: Context Explosion

We first describe briefly how object-sensitive pointer analysis models context-sensitivity in terms of object-sensitivity (Section 2.2.1). We then use an example to highlight the context explosion problem inherent in all existing object-sensitive pointer analysis algorithms (Section 2.2.2).

2.2.1 Object Sensitivity. We use a simple example given in Figure 2 to explain how the (calling) contexts of a method are modeled object-sensitively for now and how our context-debloating approach works later. In lines 7–11, we define class `A`, which has a field `f` and its setter and getter methods. In lines 12–37, we define class `B`, which has a field `g`, a constructor, and three regular methods (`createA()`, `foo()` and `bar()`). The constructor allocates an instance of `A`, denoted `A2`, and initializes `g` to point to `A1`, which is allocated also as an instance of `A` in `createA()` and made to point to `A2` (via its field `f`). In `foo()` (`bar()`) of class `B`, an instance of `java.lang.Object`, denoted `O1` (`O2`), is created. Later, `O1` (`O2`) is first stored into `A2.f` and then loaded into `v1` (`v2`) via `setF()` and `getF()`, respectively. In `main()`, two instances of `B`, denoted `B1` and `B2`, are created and used as the receivers to invoke `foo()` and `bar()`, respectively.

In a context-insensitive Andersen's analysis [1, 21], every method is analyzed only once under an empty context, [ ]. In this article, we write $\overline{pts}(v)$ to represent the points-to set of a variable v thus computed. As illustrated in Figure 3(a), `O1` and `O2` are merged at `o` (line 9) and will later flow spuriously to `v2` and `v1`, respectively. As a result, we have $\overline{pts}$(`v1`) = $\overline{pts}$(`v2`) = {`O1`, `O2`}.

In a k-object-sensitive pointer analysis (kOBJ), denoted PTA, the calling contexts of a method are distinguished by its receiver objects, with each being abstracted by its k-most-recent allocation sites [34, 35]. We write $pts_{PTA}(v, c)$ to represent the points-to set of a variable v thus computed under a given context c. In the case of 2OBJ (i.e., kOBJ with $k = 2$), `setF()` (`getF()`) will be analyzed differently for its two invocations in lines 28 and 35 (lines 29 and 36) under two different contexts, [`A2`, `B1`] and [`A2`, `B2`]. As a result, `O1` (created under context [`B1`]) and `O2` (created under context [`B2`]) will flow along two separate paths to `v1` and `v2`, respectively, as shown in Figure 3(b). This time,

```
 1 void main() {
 2   B b1 = new B(); // B1
 3   b1.foo();
 4   B b2 = new B(); // B2
 5   b2.bar();
 6 }
 7 class A {
 8   Object f;
 9   void setF(Object o) { this.f = o; }
10   Object getF() { return this.f; }
11 }
12 class B {
13   A g;
14   B() {
15     A a2 = new A(); // A2
16     A a3 = this.createA(a2);
17     this.g = a3;
18   }
19   A createA(A p) {
20     A a1 = new A(); // A1
21     a1.f = p;
22     return a1;
23   }
24   void foo() {
25     Object o1 = new Object(); // O1
26     A t1 = this.g;
27     A a1 = t1.f;
28     a1.setF(o1);
29     Object v1 = a1.getF();
30   }
31   void bar() {
32     Object o2 = new Object(); // O2
33     A t2 = this.g;
34     A a2 = t2.f;
35     a2.setF(o2);
36     Object v2 = a2.getF();
37 }}
```

Fig. 2. An example for illustrating object sensitivity as a context abstraction.

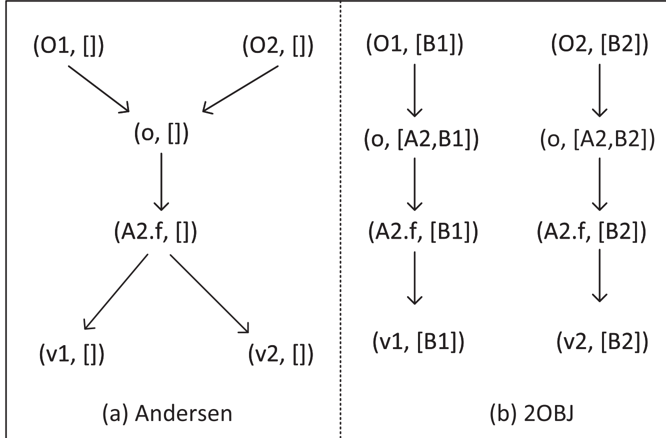


Fig. 3. Computing the points-to information for v1 and v2 in the program given in Figure 2 by applying (context-insensitive) Andersen's pointer analysis and 2OBJ (2-object-sensitive pointer analysis).

$pts_{2OBJ}(\texttt{v1}, [\texttt{B1}]) = \{(\texttt{O1}, [\texttt{B1}])\}$ and $pts_{2OBJ}(\texttt{v2}, [\texttt{B2}]) = \{(\texttt{O2}, [\texttt{B2}])\}$, without the spurious points-to information introduced by Andersen's analysis previously.

In general, when a method m is analyzed under a (calling or method) context $[o_1, \ldots, o_k]$, o_1 is a receiver object of m, and o_i is a receiver object of a method where o_{i-1} is allocated, and thus known as an *allocator (object)* of o_{i-1}, where $1 < i \leqslant k$. Therefore, any object o_0 that is allocated in m is identified as $(o_0, [o_1, \ldots, o_{k-1}])$, where $[o_1, \ldots, o_{k-1}]$ is known as the *heap context* of o_0.

```
1  class A {
2    Object f;
3    void setF(Object o) { this.f = o; }
4    Object getF() { return this.f; }
5  }
6  class B {
7    A g;
8    B() {
9      A a2 = new A(); // A2
10     A a3 = this.createA(a2);
11     this.g = a3;
12   }
13   A createA(A p) {
14     A a1 = new A(); // A1
15     a1.f = p;
16     return a1;
17   }
18   void foo() {
19     Object o1 = new Object(); // O1
20     A t1 = this.g;
21     A a1 = t1.f;
22     a1.setF(o1);
23     Object v1 = a1.getF();
24   }
25   void bar() {
26     Object o2 = new Object(); // O2
27     A t2 = this.g;
28     A a2 = t2.f;
29     a2.setF(o2);
30     Object v2 = a2.getF();
31 }}
32 class D {}
33 class C {
34   void foo_{i,j}() { // j < 2^{i-1}
35     C c_{i,j} = new C(); // C_{i,j}
36     c_{i,j}.foo_{i-1,j/2}();
37   }
38   D bar_{i,j}(D d) { // 2^{i-1} ≤ j
39     C c_{i,j} = new C(); // C_{i,j}
40     c_{i,j}.bar_{i-1,j/2}(d);
41     return d;
42   }
43   void foo_{0,0}() {
44     B b3 = new B(); // B3
45     b3.foo();
46   }
47   D bar_{0,0}(D d) {
48     B b4 = new B(); // B4
49     b4.bar();
50     return d;
51 }}
52 void main() {
53   D d = new D; // D
54   C c = new C(); // C_{n,0}
55   c.foo_{n-1,0}();
56   ...
57   C c = new C(); // C_{n,2^{n-1}-1}
58   c.foo_{n-1,2^{n-2}-1}();
59   C c = new C(); // C_{n,2^{n-1}}
60   c.bar_{n-1,2^{n-2}}(d);
61   ...
62   C c = new C(); // C_{n,2^n-1}
63   c.bar_{n-1,2^{n-1}-1}(d);
64 }
```

Fig. 4. An example program for motivating our context-debloating approach, Conch, where $1 \leqslant i \leqslant n$ and $0 \leqslant j < 2^i$, by reusing classes A and B defined in lines 7–37 in the program given in Figure 2.

2.2.2 Context Explosion. We now use an example given in Figure 4, which reuses classes A and B from Figure 2 (duplicated in lines 1–31 for improving readability), to highlight the context explosion problem in existing object-sensitive pointer analysis algorithms, including *k*obj [34, 35] and recent approaches for enabling *k*obj to adopt selective context-sensitivity [11, 18, 24, 32], in analyzing real-world programs. In line 32, we define class D as an empty class. In lines 33–51, we define class C as containing a total of 2^{n+1} methods. In lines 34–42, where $0 \leqslant j < 2^{i-1}$ ($2^{i-1} \leqslant j < 2^i$), a method, $\texttt{foo}_{i,j}()$ ($\texttt{bar}_{i,j}()$), is defined, in which an object, $C_{i,j}$, is created and used as the receiver to invoke $\texttt{foo}_{i-1,\frac{j}{2}}()$ ($\texttt{bar}_{i-1,\frac{j}{2}}()$). In lines 43–51, we define $\texttt{foo}_{0,0}()$ ($\texttt{bar}_{0,0}()$), where an instance of class B, denoted B3 (B4), is created and used to invoke foo() (bar()). In main() (lines 52–64), a total of 2^n instances of class C, denoted $C_{n,j}$, where $0 \leqslant j < 2^n$, are created and used as the receiver objects to call $\texttt{foo}_{n-1,\frac{j}{2}}()$ when $j < 2^{n-1}$ and $\texttt{bar}_{n-1,\frac{j}{2}}()$ when $j \geq 2^{n-1}$.

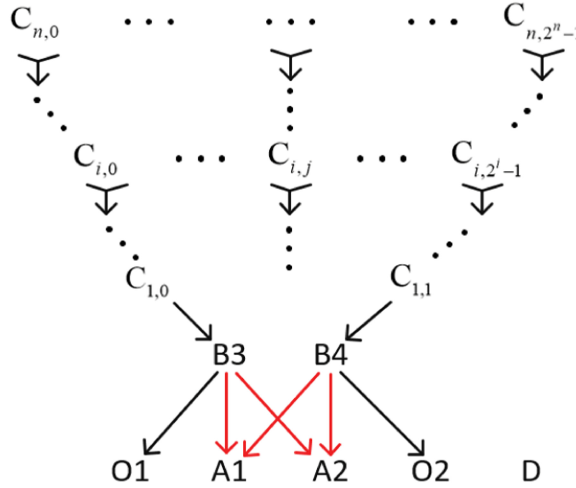


Fig. 5. The OAG for Figure 4, where only the four edges in red will remain after context debloating (so that only A1 and A2 need to be analyzed context-sensitively under [B3] and [B4]).

Figure 5 depicts the **object allocation graph** (**OAG**) [50], where a node represents an object and an edge $O \rightarrow O'$ signifies that O is an allocator of O'. For kOBJ [35, 44], the contexts of a method can be directly read off from this graph by starting from its receiver object and then retrieving the next $k-1$ objects backward. For example, the sets of contexts of foo() and bar() are $\{[\mathsf{B3}, \mathrm{C}_{1, \frac{j}{2^{k-2}}}, \ldots, \mathrm{C}_{k-2, \frac{j}{2}}, \mathrm{C}_{k-1,j}] \mid 0 \leqslant j < 2^{k-2}\}$ and $\{[\mathsf{B4}, \mathrm{C}_{1, \frac{j}{2^{k-2}}}, \ldots, \mathrm{C}_{k-2, \frac{j}{2}}, \mathrm{C}_{k-1,j}] \mid 2^{k-2} \leqslant j < 2^{k-1}\}$, respectively. Let $C_j(X) = [\mathrm{A}, X, \mathrm{C}_{1, \frac{j}{2^{k-3}}}, \ldots, \mathrm{C}_{k-3, \frac{j}{2}}, \mathrm{C}_{k-2,j}]$. Both setF() and getF() share the same set of contexts: $\{C_j(\mathsf{B3}) \mid 0 \leqslant j < 2^{k-3}\} \cup \{C_j(\mathsf{B4}) \mid 2^{k-3} \leqslant j < 2^{k-2}\}$. In practice, the number of contexts for analyzing a method can be exponential. For example, there are a total of 2^{k-2} contexts for each of the four methods, foo(), bar(), setF() and getF(), contained in classes A and B defined in lines 1–31. As k increases, each of such methods becomes exponentially expensive to analyze, consuming increasingly more memory and more analysis time.

Existing approaches for selective context-sensitivity [11, 18, 24, 32] can improve the efficiency and scalability of kOBJ. For example, ZIPPER [24], which does not preserve the precision of kOBJ, will instruct kOBJ to analyze main(), B(), foo(), bar(), and $\texttt{foo}_{i,j}$() (where $j < \frac{r^i}{2}$) context-insensitively but the remaining methods context-sensitively. As a result, the context explosion problem still remains for $\texttt{bar}_{i,j}$(), setF() and getF(). On the other hand, EAGLE [30, 32], which preserves the precision of kOBJ, is more conservative here, as it will also instruct kOBJ to analyze B(), foo() and bar() partially context-sensitively (i.e., some of their variables/objects context-sensitively).

2.3 CONCH: Context Debloating

We first explain the basic idea behind context debloating, then discuss some challenges faced for achieving it efficiently and effectively, and finally, describe our IFDS-based solution.

2.3.1 Basic Idea. We introduce a new approach to mitigating the context explosion problem. As illustrated in Figure 6, our approach, named **CONtext-dependability CHecking** (**CONCH**), aims to debloat contexts for any given object-sensitive analysis algorithm, say, *PTA* to boost its performance significantly while causing it to incur only a negligible loss in precision. The debloated

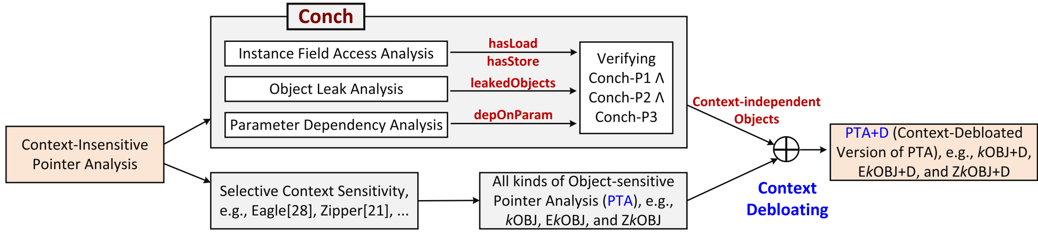


Fig. 6. The workflow of Conch in debloating contexts for an object-sensitive pointer analysis (PTA).

version of *PTA* is referred to as *PTA+D*. *PTA* can be *k*obj or any of its incarnations developed for supporting selective context-sensitivity, such as E*k*obj configured by Eagle [32] or Z*k*obj configured by Zipper [24], guided often by Andersen's context-insensitive pointer analysis [1]. Due to context debloating, the debloated counterparts of *k*obj, E*k*obj and Z*k*obj are obtained as *k*obj+D, E*k*obj+D and Z*k*obj+D, respectively. Therefore, our work is both orthogonal and complementary to the prior work on selective context-sensitivity, including Eagle [32], Zipper [24], and others [11, 18]. To debloat contexts for *PTA* and thus obtain *PTA+D*, Conch will divide the objects in a program into context-dependent and context-independent objects, based also on the points-to information computed by Andersen's pointer analysis. Then Conch will alleviate the context explosion problem suffered by *PTA* by instructing *PTA+D* to proceed exactly the same as *PTA* except that *PTA+D* will now analyze all the identified context-independent objects context-insensitively if *PTA* has not been designed to do so. We will discuss all the analysis phases used in Conch in Section 4.

Let us apply Conch to *k*obj with respect to our motivating example given Figure 4. By its design, *k*obj will analyze all its objects context-sensitively. However, as `A1` and `A2` are the only two context-dependent objects, analyzing any other object context-sensitively will cost an exponential increase in analysis time without achieving any precision benefit. For all these objects, we can apply context debloating to *k*obj by instructing it to analyze only `A1` and `A2` context-sensitively. To illustrate this with respect to the OAG given in Figure 5, we will end up removing effectively all the allocators of every context-independent object so that the exponential growth of contexts for the object is avoided completely. In the end, the four edges highlighted in red will remain, as `A1` and `A2` are the only two context-dependent objects. This implies that *k*obj will now analyze only `createA()` context-sensitively (under `[B3]` and `[B4]`) and `setF()` and `getF()` context-sensitively (under `[A2, B3]` and `[A2, B4]`), but all the other methods are context-insensitively. For this particular example, debloating contexts can help *k*obj (and also all its variants) reduce their analysis times and memory consumption significantly without losing any precision.

In practice, Conch can be deployed straightforwardly to improve the efficiency and scalability of object-sensitive pointer analysis tools. Let there be a set of n such tools, $PTA_1, \ldots, PTA_n$, which may include *k*obj or its incarnations for supporting selective context-sensitivity prescribed by pre-analyses such as Eagle and Zipper. Suppose that the developer has decided to select PTA_i (for some i) to analyze a particular set of programs, P, since it satisfies some particular efficiency-precision tradeoff required. Let $PTA_i + D$ be the debloated version of PTA_i obtained by Conch. By design, $PTA_i + D$ is expected to run much faster than PTA_i while offering nearly the same precision. Therefore, the developer can always use $PTA_i + D$ as a better alternative to PTA_i in analyzing P.

2.3.2 Challenges. To debloat contexts for object-sensitive pointer analysis, we must find context-dependent (or equivalently, context-independent) objects in a given program. Recently, the following two necessary conditions, denoted C1 and C2 (rephrased from the two field-sensitive

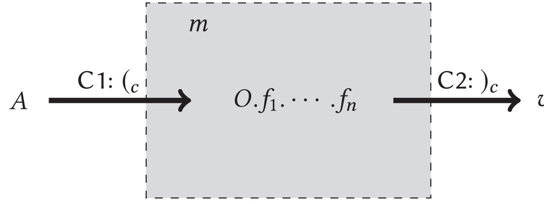


Fig. 7. Two necessary conditions C1 and C2 for an object O allocated in method m to be context-dependent.

conditions, FS1 and FS2, given in [32]), which are practically but not theoretically sufficient for real-world programs, have been proposed for determining the context-dependability of an object O allocated in a method m based on CFL reachability. These two conditions, which are illustrated in Figure 7, require us to check the existence of a write into and a read from the same access path $O.f_1.\cdots.f_n$ context-sensitively (with the write and read accesses happening frequently outside m) [30–32]:

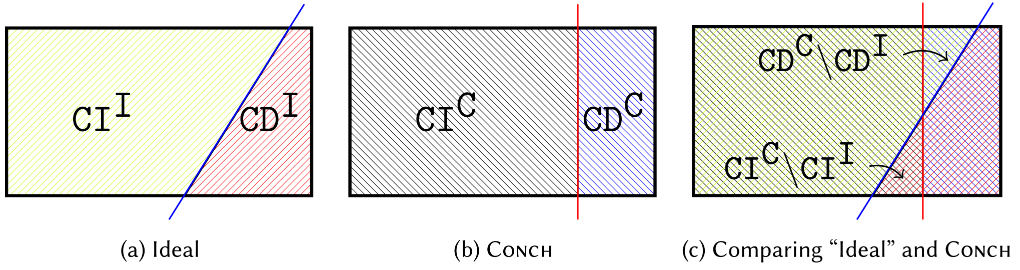
C1. $A \xrightarrow{(_c} O.f_1.\cdots.f_n$: there exists an object A that flows into m from outside and ends up being stored later into $O.f_1.\cdots.f_n$ under a calling context c of m, and

C2. $O.f_1.\cdots.f_n \xrightarrow{)_c} v$: there exists a load of $O.f_1.\cdots.f_n$ flowing into a variable v outside m under also the same calling context c.

where f_i may be either a real Java field or a special field (which is a parameter or the return variable of a method modeled conceptually as a field of its receiver object on which the method is called) [30, 32]. For C1 and C2, context matching, which is indicated by "$(_c$" and "$)_c$", is formulated by solving the standard balanced parentheses problem [40]. If O is context-dependent, then C1∧C2 must hold, which implies the existence of both a context-sensitive value flow from an object A outside m into $O.f_1.\cdots.f_n$ and of a context-sensitive value flow from $O.f_1.\cdots.f_n$ into a variable v outside m (as illustrated graphically in Figure 7). Conversely, if C1 ∧ C2 holds, then O is usually context-dependent in real-world programs (even though C1 ∧ C2 is not sufficient). In this case, if O is considered as being context-independent, then different objects A (if they exist) that flow into $O.f_1.\cdots.f_n$ under different calling contexts of m will be conflated, causing them to flow into different variables v (if they exist) spuriously. Therefore, for all practical purposes (by assuming that a method is usually called more than once in the program), O can be regarded as being context-independent if and only if ¬(C1 ∧ C2) holds.

Suppose we want to apply context debloating to kOBJ. Unfortunately, checking whether O is context-dependent or not by verifying C1∧C2 precisely is undecidable [38], as this ideal approach will require us to solve kOBJ fully context-sensitively (with $k = \infty$) for a program. In addition, weakening C1 ∧ C2, i.e., strengthening ¬(C1 ∧ C2), as done by EAGLE [30, 32] directly, will make many of the value-flows like those depicted in Figure 7 spurious, thereby over-approximating unduly the number of context-dependent objects, i.e., under-approximating unduly the number of context-independent objects found in a program, even though such a conservative approach will always preserve the precision of kOBJ. On the other hand, approximating C1 ∧ C2 heuristically, as done by ZIPPER [24] indirectly (as C1 and C2 were introduced one year later), may cause many context-dependent objects to be mis-classified as being context-independent, and vice versa, even though such a heuristic approach may speed up kOBJ at often a noticeable precision loss (e.g., an average loss of 10.1% for the *#fail-cast* client in our evaluation discussed in Section 5.1).

2.3.3 Our Solution. To identify context-dependent objects efficiently and effectively, our guiding principles are to approximate the two aforementioned necessary conditions, C1 and C2, with



(a) Ideal (b) Conch (c) Comparing "Ideal" and Conch

Fig. 8. Approximating C1 ∧ C2 in the ideal case by Conch-P1 ∧ Conch-P2 ∧ Conch-P3 in Conch.

a set of three conditions, Conch-P1, Conch-P2 and Conch-P3, so that these three conditions are efficiently verifiable and mostly necessary in practice. In order for these three conditions to be verified efficiently for a program (as described in Section 4), we will see that Conch-P1 can be checked linearly in terms of the number of objects in the program according to $\overline{pts}$. In order for each of the remaining two conditions (Conch-P2 and Conch-P3) to be checked efficiently, we will introduce an IFDS-based solution, which runs linearly in terms of the number of PAG edges [21] in the program. To ensure that Conch-P1 ∧ Conch-P2 ∧ Conch-P3 approximates C1 ∧ C2 reasonably accurately (by avoiding many spurious C1 ∧ C2-satisfying value flows introduced by Eagle [30, 32]), as illustrated in Figure 8, our key insight is to develop these three conditions based on three key observations governing how objects are used in real-world object-oriented programs, with one condition per observation. In Figure 8(a), CI^I and CD^I represent the sets of context-independent and context-dependent objects found for a program, respectively, according to C1 ∧ C2 in the ideal situation. In Figure 8(b), CI^C and CD^C represent their counterparts found by Conch according to Conch-P1 ∧ Conch-P2 ∧ Conch-P3. To ensure that Conch-P1 ∧ Conch-P2 ∧ Conch-P3 approximates C1 ∧ C2 reasonably accurately (i.e., enable Conch to accelerate *k*obj at a little loss of precision), as illustrated in Figure 8(c), we will aim at minimizing both (1) $|CD^C \setminus CD^I|$ (in order to boost its performance by minimizing the number of context-independent objects in CI^I that are mis-classified as being context-dependent) and (2) $|CI^C \setminus CI^I|$ (in order to reduce its precision loss by minimizing the number of context-dependent objects in CD^I that are mis-classified as being context-independent). It should be pointed out that different objects in a program, once analyzed context-sensitively or context-insensitively, may impact the efficiency and/or precision of the program differently, but how to quantify this is an open problem [32].

Like the prior work on selective context-sensitivity [11, 18, 24], Conch also relies on the points-to information, $\overline{pts}$, pre-computed (context-insensitively) by Andersen's analysis.

We are now ready to introduce Conch-P1, Conch-P2, and Conch-P3. For each condition, we will describe what the condition is, explain the intuition behind it, and illustrate it with some examples. Note that Conch-P3 is a disjunct of two sub-conditions: Conch-P3 := Conch-P3a ∨ Conch-P3b.

Observation 1 (Conch-P1). *A context-dependent object O often has at least one pointer field $O.f$ that is both written into (via a store $x.f = \cdots$) and read from (via a load $\cdots = y.f$) in the program, where $O \in \overline{pts}(x)$ and $O \in \overline{pts}(y)$, i.e., x and y are aliases according to $\overline{pts}$.*

Intuitively, if O is context-dependent, then there will be a pair of load and store statements into one of O's pointer fields. If such a pair is absent, C1 ∧ C2 will usually fail to hold, since one of the two value flows needed for establishing C1 and C2, as illustrated in Figure 7, will likely be absent.

Figure 9 gives an example where `O`, an array object created in constructor `ArrayList()` in line 5, is context-dependent. As described in Section 3, an array object is modeled monolithically

```
1  class A {}                                11    return this.data[0];
2  class ArrayList {                         12 }}
3    Object[] data;                          13 void main() {
4      ArrayList() {                         14   List l1 = new ArrayList(); // L1
5        this.data = new Object[10]; // O    15   l1.add(new A()); // A1
6      }                                     16   Object v1 = l1.get();
7    void add(Object elem) {                 17   List l2 = new ArrayList(); // L2
8      this.data[0] = elem;                  18   l2.add(new A()); // A2
9    }                                       19   Object v2 = l2.get();
10   Object get() {                          20 }
```

Fig. 9. A context-dependent object satisfying Conch-P1 in a code snippet abstracted from JDK.

```
1 void main() {                              8      return q;
2   A a = new A(); // A                      9  }}
3   Object o = new Object(); // O            10 class A {
4   Object v = a.wrapId(o);                  11   Object wrapId(Object p) {
5 }                                          12     B b = new B();  // B
6 class B {                                  13     return b.id(p);
7   Object id(Object q) {                    14 }}
```

Fig. 10. A context-dependent object B violating Conch-P1.

(i.e., element-insensitively) so that all its element accesses are interpreted as happening to a special field, denoted *arr*. For the array object O with its allocating method m = `ArrayList`, we will see that C1 ∧ C2, as illustrated in Figure 7, holds. It is easy to see that O satisfies Conch-P1 due to the store `this.data.arr = elem`, i.e., `O.arr = elem` in line 8 and the load from `this.data.arr`, i.e., `O.arr` in line 11. When O is analyzed context-sensitively under [Li] based on CFL reachability, Ai will flow into `O.arr` due to this store (so that C1 is satisfied) and `O.arr` will flow into vi due to this load (so that C2 is satisfied), where $1 \leqslant i \leqslant 2$. As a result, v1 will point to A1 and v2 will point to A2, as expected. If O is analyzed context-insensitively, v1 (v2) will also point to A2 (A1) spuriously.

There can be rare cases, as illustrated in Figure 10, where Conch-P1 does not hold for some context-dependent objects, such as the object B allocated in line 12. Under object-sensitivity [34, 35], the object O that is pointed to by p in line 11 is first written into B.q due to the call to `id()` in line 13 and then returned and stored into v in line 4. As discussed in Section 2.3.2, q is therefore considered as a special field of B. Such cases are rare, as Conch is observed to lose little precision in real-world object-oriented programs during our extensive evaluation described later in Section 5.

Observation 2 (Conch-P2). *A context-dependent object O, pointed to by a variable or a pointer field of some object according to $\overline{pts}$, usually flows out of its* allocating method *(where O is created).*

Intuitively, once created in such a factory-like allocating method, O will likely be subject to different field accesses (to $O.f_1.\cdots.f_n$) under different calling contexts. If O does not flow out of its allocating method, then C2 will usually fail to hold, i.e., its corresponding value-flow illustrated in Figure 7 will likely be absent in real-world object-oriented programs. For the program given in Figure 9, the array object O allocated in line 5 obviously satisfies Conch-P2. As discussed earlier, O is context-dependent, since Ai will flow into `this.data[0]`, i.e., `O.arr` due to the store in line 8 (so that C1 is satisfied) and `this.data[0]`, i.e., `O.arr` will flow into vi due to the load in line 11

```
1 Vector(int size) {
2   this.elems = new Object[size];
3 }
```

(a) Case 1: from Vector

```
1 void reconstitutionPut(Entry[] entries, K key, V value) {
2   Entry entry = new Entry(key, value);
3   entries[key.hashcode] = entry;
4 }
```

(b) Case 2: from Hashtable

```
1 Iterator iterator() {
2   return new KeyIterator();
3 }
```

(c) Case 3: from HashMap

```
1 StreamTokenizer(InputStream in) {
2     this.input = in;
3 }
4 void SunJCE_e_a(...) {
5   BufferedInputStream bin = new BufferedInputStream();
6   this.f = new StreamTokenizer(bin);
7 }
```

(d) Case 4: from SunJCE_e

Fig. 11. Four common cases abstracted from JDK for satisfying Conch-P2.

(so that C2 is satisfied), where $1 \leqslant i \leqslant 2$. In rare cases, such as the one illustrated in Figure 10, B, which violates Conch-P2, is still context-dependent for the reasons explained earlier.

Figure 11 gives four representative cases abstracted from the JDK where Conch-P2 holds. In Figure 11(a), the array object created flows out of the constructor via a store into its `this` variable. In Figure 11(b), the `Entry` object created flows out of `reconstitutionPut()` via its parameter `entries`. In Figure 11(c), the `KeyIterator` object created flows out of `iterator()` directly via a return statement. Finally, in Figure 11(d), we have a slightly more complex case. The `BufferedInputStream` object created flows out of its allocating method, `SunJCE_e_a()`, as it is stored into the `input` field of a `StreamTokenizer` object, which flows out of the allocating method via a store into its `this` variable. In general, the objects that do not flow out of their allocating methods are usually context-independent as they are often created and used locally.

Observation 3 (Conch-P3 := Conch-P3a ∨ Conch-P3b). *A context-dependent object O tends to be involved in a store statement $x.f = y$ contained in a method m', where $O \in \overline{pts}(x)$. Let m be the method that allocates O if m' happens to be a constructor (i.e., the constructor called for creating O) and m' otherwise. Then, for the following two sub-conditions given, at least one tends to hold:*

Conch-P3a. *y is data-dependent on a formal parameter of m.*
Conch-P3b. *y points to a context-dependent object.*

Conch-P3a says that an object flows into method m directly (via one of its parameters) and ends up being stored into $O.f.g.\cdots$. Conch-P3b says that an object flows into method m indirectly (via another context-dependent object) and ends up being stored into $O.f.g.\cdots$. Intuitively, if neither

```
1 ArrayList() {
2   this.elems = new Object[5];
3 }
4 void set(int idx, E e) {
5   this.elems[idx] = e;
6 }
```

(a) Case 1: from ArrayList

```
1 void addEntry(int idx, K k, V v) {
2   this.table[idx] = new Entry(k,v);
3 }
4 Entry(K k, V v) {
5   this.key = k;
6   this.value = v;
7 }
```

(b) Case 2: from HashMap

```
1 HashSet() {
2   this.map = new HashMap();
3 }
4 HashMap(...) {
5   this.table = new Entry[10];
6 }
7 V get(K key) {
8   Entry entry = this.table[key.hashcode];
9   return entry.value;
10 }
11 void transfer(Entry[] entries) {
12     this.table[0] = entries[0];
13 }
```

(c) Case 3: from HashSet and HashMap

Fig. 12. Three common cases abstracted from JDK for satisfying Conch-P3.

Conch-P3a nor Conch-P3b holds, then C1 will usually fail to hold, i.e., its corresponding value-flow illustrated in Figure 7 will likely be absent in real-world object-oriented programs. Note that O is allocated in m when $m \neq m'$ but may or may not be allocated in m when $m = m'$. Therefore, we can distinguish two cases, depending on whether O is allocated in m or not:

- **Case A. O is Allocated in m.** This happens when $m \neq m'$ (implying that O is allocated in m by the definition of m) or $m = m'$ with O happening to be allocated in m.
- **Case B. O Is Not Allocated in m.** This happens when $m = m'$ with O happening not to be allocated in m, i.e., O being allocated in a method that is different from m.

In Case A, C1 will likely hold, i.e., its corresponding value-flow illustrated in Figure 7 will likely exist when either Conch-P3a or Conch-P3b is true. In Case B, where m contains $x.f = y$, such a value-flow that reaches $O.f.g.\cdots$ contained in m will likely also reach $O.f.g.\cdots$ contained in its allocating method (based on CFL reachability) since `x` points to `O`, so that C1 will also likely hold.

Figure 12 gives three representative cases abstracted from the JDK where Conch-P3 holds. In Figure 12(a), O is the `Object[]` object allocated in line 2, and $x.f = y$ is `this.elems[idx] = e`

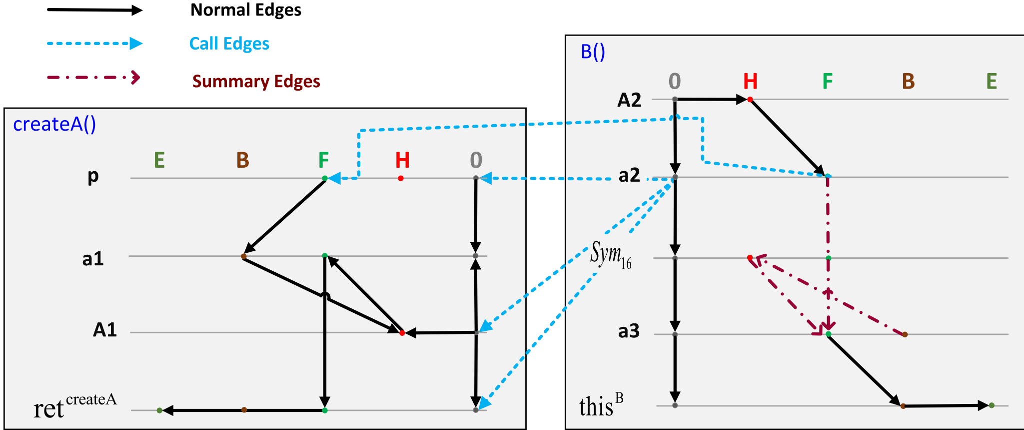


Fig. 13. An IFDS-based object reachability analysis performed by Conch on the PAG of the motivating example given in Figure 4 to verify Conch-P2 for its objects (with only A1 and A2 shown to avoid cluttering).

contained in method m' = `set()` in line 5, which is modeled as `this.elems.arr = e`, where *arr* is a special field introduced to represent all the elements of an array (Section 3). Here, x is `this.elems`, f is *arr*, and y is `e`. As `set()` is not a constructor, we have $m = m'$ = `set()`. In this first case, y, i.e., `e` satisfies Conch-P3a trivially. In Figure 12(b), O is the `Entry` object allocated in line 2 (with `addEntry()` as its allocating method) and $x.f = y$ can be considered as representing either `this.key = k` (line 5) or `this.value = v` (line 6). As `Entry()` is a constructor, we have m' = `Entry()` and m = `addEntry()`. In this second case, y, which can be either `k` or `v`, satisfies Conch-P3a trivially. Finally, in Figure 12(c), we consider a slightly more complex case, where O is the `HashMap` object allocated in `HashSet()` (line 2), $x.f = y$ is `this.table = new Entry[10]`, m' = `HashMap()`, and m = `HashSet()` (since `HashMap()` is a constructor). We find that y, i.e., `new Entry[10]` is context-dependent since (1) Conch-P1 holds (due to the store in line 12 and the load in line 8), (2) Conch-P2 holds (due to line 5), and (3) Conch-P3a holds (due to line 12, where `entries` is a parameter of `transfer()`). As a result, the `HashMap` object allocated in line 2 is also considered as being context-dependent by Conch-P3b. In our context-debloating approach (Section 4), the circular dependencies on context-dependability are solved iteratively.

To verify these three conditions for a program efficiently (as described in Section 4), we will see that Conch-P1 can be checked linearly in terms of the number of objects in the program according to $\overline{pts}$. To check each of the remaining two conditions (Conch-P2 and Conch-P3), we introduce an IFDS-based solution, which runs linearly in terms of the number of PAG edges [21] in the program.

Finally, let us apply context debloating to *k*obj in analyzing our motivating example given in Figure 4. For Conch-P1, we will see below how to check it directly according to $\overline{pts}$. For Conch-P2, we will check it by performing an IFDS-based object reachability analysis on the PAG of the example program, as illustrated in Figure 13. For Conch-P3, we will check Conch-P3a by performing an IFDS-based object reachability analysis on the same PAG, as illustrated in Figure 14. We will revisit Figures 13 and 14 when we introduce our Conch approach in Section 4.

For this example, Conch will identify `A1` and `A2` as the only two context-dependent objects. As shown in Figure 4, `A1` is allocated in line 14 when `createA()` is called and `A2` is allocated in line 9 when the constructor `B()` is called. `A1` is found to be context-dependent as it satisfies Conch-P1 ∧ Conch-P2 ∧ Conch-P3: (1) `A1` has a pointer field `f`, which has a write in line 15 and two reads in lines 21 and 28 (Conch-P1), (2) `A1` can flow out of `createA()` via its return statement in line 16 (Conch-P2), captured by $\langle \texttt{A1}, H\rangle \rightarrow \langle \texttt{ret}^{\texttt{createA}}, E\rangle$ in Figure 13, and (3) `p` is stored into

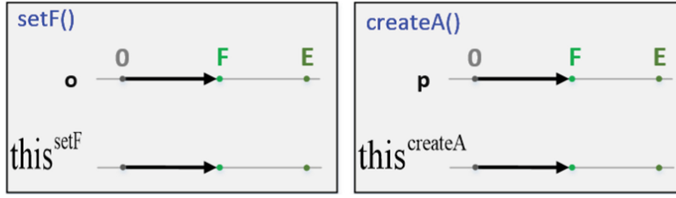


Fig. 14. An IFDS-based object reachability analysis performed by Conch on the PAG of the motivating example given in Figure 4 to verify Conch-P3a for its objects (with only the four method parameters relevant to A1 and A2, i.e., o and this$^{\text{setF}}$ of setF() and this$^{\text{createA}}$ and p of createA(), being shown).

A1.f in line 15, where p points is a parameter by itself (Conch-P3a), captured by $\langle \text{p}, F\rangle \rightarrow \langle \text{p}, F\rangle$ in Figure 14. Similarly, A2 is context-dependent as it also satisfies Conch-P1 ∧ Conch-P2 ∧ Conch-P3: (1) A2 has a pointer field f, which has a write and a read in lines 3 and 4, respectively (Conch-P1), (2) A2 can flow out of B() via the store statement in line 11 indirectly (Conch-P2), captured by $\langle \text{A2}, H\rangle \rightarrow \langle \text{this}^{\text{B}}, E\rangle$ in Figure 13, and (3) o is stored into A2.f in line 3, where o is a parameter of setF() (Conch-P3a), captured by $\langle \text{o}, F\rangle \rightarrow \langle \text{o}, F\rangle$ in Figure 14.

Let us now consider B3 and B4 allocated in Figure 4. Both are context-independent as they both satisfy Conch-P1 (due to a store into B3.g and B4.g in B() (line 11) as well as a load from B3.g in foo() (line 20) and a load from B4.g in bar() (line 27) in Figure 2) and Conch-P3 (due to the existence of this.g = a3 in line 11, where a3 points to A1, which is context-dependent) but violate Conch-P2 (as B3/B4 does not flow out of its allocating method $\text{foo}_{0,0}$()/$\text{bar}_{0,0}$(), where B3/B4 is allocated). Finally, all the other objects are context-independent as they do not contain any pointer fields and are used only locally, failing to satisfy any of the three conditions stated.

3 CONTEXT DEBLOATING FOR OBJECT-SENSITIVE POINTER ANALYSIS

As illustrated in Figure 6, Conch can be used to debloat contexts for any object-sensitive pointer analysis algorithm *PTA* by simply instructing it to analyze every context-independent object identified by Conch context-insensitively if it is not designed to do so. We formalize context debloating for *k*obj (i.e., by assuming that *PTA* = *k*obj). Note that *k*obj is context-sensitive, with field-sensitivity assumed by default in modern pointer analysis frameworks. Conch can be used similarly to debloat contexts for any variant of *k*obj. In this section, we first review the classic algorithm for *k*obj (Section 3.1) and then adapt it to support context debloating (Section 3.2).

3.1 *k*obj

We describe *k*obj [34, 35, 44] by considering a simplified subset of Java, with five types of labeled statements listed in Table 1. Note that "x = **new** *T*(...)" is modeled as "x = **new** *T*; x.⟨init⟩(...)", where ⟨init⟩(...) is the corresponding constructor invoked. The control flow statements are irrelevant since *k*obj is context-sensitive but flow-insensitive. As *k*obj is array-insensitive, loads and stores to the elements of an array are modeled by collapsing all the elements into a special field *arr* of the array. Every method is assumed to have one return statement "**return** *ret*", where *ret* is known as its *return variable*. Section 5 discusses how to handle static method calls and other complex language features such as exceptions, reflection, and native code.

*k*obj makes use of a few domains, $\mathbb{V}$, $\mathbb{H}$, $\mathbb{M}$, $\mathbb{F}$, and $\mathbb{L}$, which represent the sets of program variables, heap objects (identified by their labels), methods, field names, and statements (identified also by their labels), respectively. We use $\mathbb{C} = \mathbb{H}^*$ as the universe of contexts. Given a context $ctx = [e_1, \ldots, e_n] \in \mathbb{C}$ and a context element $e \in \mathbb{H}$, we write $e :: ctx$ for $[e, e_1, \ldots, e_n]$ and $\lceil ctx \rceil_k$ for $[e_1, \ldots, e_k]$. Given these notations, the following auxiliary functions are defined:

Table 1. Five Types of Statements Analyzed by *k*OBJ

Kind	Statement	Kind	Statement
NEW	$l : x = \textbf{new}\ T$	ASSIGN	$l : x = y$
STORE	$l : x.f = y$	LOAD	$l : x = y.f$
CALL	$l : x = a_0.f(a_1, \ldots, a_r)$		

— *methodOf* : $\mathbb{L} \rightarrow \mathbb{M}$
— *methodCtx* : $\mathbb{M} \rightarrow \wp(\mathbb{C})$
— *dispatch* : $\mathbb{L} \times \mathbb{H} \rightarrow \mathbb{M}$
— *pts* : $(\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C} \rightarrow \wp(\mathbb{H} \times \mathbb{C})$

where *methodOf* gives the containing method of a statement, *methodCtx* maintains the contexts used for analyzing a method, *dispatch* resolves a virtual call to a target method, and *pts* records the points-to information found context-sensitively for a variable or an object's field.

Figure 15 gives the five rules used by *k*OBJ for analyzing the five kinds of statements listed in Table 1. In [NEW], $O_l \in H$ is an abstract heap object created from the allocation site at l, identified by its heap context *hctx*. Rules [ASSIGN], [STORE], and [LOAD] are applied to handle assignments, stores, and loads, respectively, in the standard manner. In [CALL], a call to an instance method $x = a_0.f(a_1, \ldots, a_r)$ is analyzed. In this article, we write $this^{m'}$, $p_i^{m'}$ and $ret^{m'}$ for the "this" variable, ith parameter and return variable of m', respectively, where m' is a target method dispatched. Frequently, we also write $p_0^{m'}$ for $this^{m'}$. In the conclusion of this rule, $ctx' \in methodCtx(m')$ reveals how the contexts of a method are maintained. Initially, *methodCtx*("`main`") = {[ ]}.

3.2 Context Debloating

To debloat contexts for *k*OBJ or any other variant of *k*OBJ, we assume that $\mathcal{I}$ represents the set of context-independent objects found by CONCH. Therefore, the objects in $\mathbb{H} \setminus \mathcal{I}$ are context-dependent. To modify *k*OBJ to support context debloating, we simply need to replace [NEW] given in Figure 15 by [NEW+D] given in Figure 16. For a context-dependent object, we proceed identically as before. For a context-independent object, we will instruct *k*OBJ to analyze it context-insensitively now since *k*OBJ would otherwise analyze it context-sensitively. This means that we will no longer distinguish it under its different allocators by setting its heap context as $hctx = [\,]$, thereby eliminating completely the context explosion problem that would otherwise have occurred when it is used as a receiver object of an invoked method, as illustrated in Figure 5 for our motivating example. To debloat contexts for all incarnations of *k*OBJ such as those supporting selective context-sensitivity prescribed by EAGLE [30, 32] and ZIPPER [24], we can make similar modifications.

CONCH is conceptually simple and algorithmically easy to plug into any existing object-sensitive pointer analysis, and practically effective as validated during our extensive evaluation.

4 CONCH: DETERMINING THE CONTEXT-DEPENDENCIES OF OBJECTS

CONCH, with its IFDS-based algorithm given in Algorithm 1 and its workflow given earlier in Figure 6, is designed to verify CONCH-P1 ∧ CONCH-P2 ∧ CONCH-P3 efficiently in separating the context-dependent objects from the context-independent objects in a given program, by leveraging the points-to information $\overline{pts}$ pre-computed by Andersen's analysis [1] (the context-insensitive version of Figure 15). According to our algorithm, an object is classified as being context-dependent (context-independent) if and only if CONCH-P1∧CONCH-P2∧CONCH-P3 (¬(CONCH-P1∧CONCH-P2∧CONCH-P3)) holds. To check CONCH-P1, we only require $\overline{pts}$. To check both CONCH-P2 and CONCH-P3a, we solve an IFDS-based object reachability problem for each. To check CONCH-P3b, we

$$\frac{l : x = \text{new } T \quad m = methodOf(l) \quad ctx \in methodCtx(m) \quad hctx = \lceil ctx \rceil_{k-1}}{(O_l, hctx) \in pts(x, ctx)} \quad \text{[New]}$$

$$\frac{l : x = y \quad m = methodOf(l) \quad ctx \in methodCtx(m)}{pts(y, ctx) \subseteq pts(x, ctx)} \quad \text{[Assign]}$$

$$\frac{l : x.f = y \quad m = methodOf(l) \quad ctx \in methodCtx(m) \quad (O, hctx) \in pts(x, ctx)}{pts(y, ctx) \subseteq pts(O.f, hctx)} \quad \text{[Store]}$$

$$\frac{l : x = y.f \quad m = methodOf(l) \quad ctx \in methodCtx(m) \quad (O, hctx) \in pts(y, ctx)}{pts(O.f, hctx) \subseteq pts(x, ctx)} \quad \text{[Load]}$$

$$\frac{\begin{array}{c} l : x = a_0.f(a_1, ..., a_r) \quad m = methodOf(l) \\ ctx \in methodCtx(m) \quad (O, hctx) \in pts(a_0, ctx) \\ m' = dispatch(l, O) \quad ctx' = O :: hctx \end{array}}{\begin{array}{c} ctx' \in methodCtx(m') \quad (O, hctx) \in pts(this^{m'}, ctx') \\ \forall i \in [1, r] : pts(a_i, ctx) \subseteq pts(p_i^{m'}, ctx') \quad pts(ret^{m'}, ctx') \subseteq pts(x, ctx) \end{array}} \quad \text{[Call]}$$

Fig. 15. Rules for $k$obj.

$$\frac{\begin{array}{c} l : x = \text{new } T \quad m = methodOf(l) \quad ctx \in methodCtx(m) \\ hctx = \begin{cases} [\,] & \text{if } O_l \in \mathcal{I} \\ \lceil ctx \rceil_{k-1} & \text{if } O_l \in \mathbb{H} \setminus \mathcal{I} \end{cases} \end{array}}{(O_l, hctx) \in pts(x, ctx)} \quad \text{[New+D]}$$

Fig. 16. Adapting [New] to support context debloating for $k$obj.

exploit the transitivity of context-dependability across the objects in the program based on their field points-to information. As Conch-P1 $\wedge$ Conch-P2 $\wedge$ Conch-P3 are constructed to determine the context-dependability of an object necessarily but not sufficiently, we may mis-classify some context-independent objects as being context-dependent (by erring on the side of preserving precision). As Conch-P1 $\wedge$ Conch-P2 $\wedge$ Conch-P3 are mostly but not strictly necessary in real-world object-oriented programs (Figure 10), we may occasionally mis-classify some context-dependent objects as being context-independent (at a small loss of precision), as illustrated in Figure 8.

We first give a high-level overview of Algorithm 1 and then discuss how to verify the three conditions in Conch-P1 $\wedge$ Conch-P2 $\wedge$ Conch-P3 individually. Conch takes a program P as input and returns $\mathcal{I}$ as the set of context-independent objects in P for context-debloating (lines 27 and 28). Some additional notations used are in order. For a given object O, *fieldsOf*(O) denotes the set of the fields of O. In addition, $hasLoad(O, f)$ ($hasStore(O, f)$) holds if P contains a load $\cdots = x.f$ (store $x.f = \cdots$) such that $O \in \overline{pts}(x)$. In `main()`, *CI* and *CD*, which are initialized to be $\varnothing$ (line 29),

ALGORITHM 1: Conch: finding context-independent objects in P for context debloating.

```
1  Function Is_Conch-P1_Valid(O_l)
2      return ∃ f ∈ fieldsOf(O_l) : hasLoad(O_l, f) ∧ hasStore(O_l, f)
3  Function Is_Conch-P2_Valid(O_l)
4      return O_l ∈ leakedObjects
5  Function Is_Conch-P3a_Valid(O_l)
6      R(O_l) = {l' : x.f = y in P | O_l ∈ pts‾(x)}
7      for l' : x.f = y ∈ R(O_l) do
8          if methodOf(l') is a constructor of O_l then
9              m = methodOf(l)
10         else
11             m = methodOf(l')
12         if depOnParam(y, m) then
13             return true
14     return false
15 Function Check_Validity_Of_Conch-P3b(CI, CD)
16     Unknown ← ℍ \ (CI ∪ CD)
17     R(O_l) = {l' : x.f = y in P | O_l ∈ pts‾(x)}
18     changed ← true
19     while changed do
20         changed ← false
21         for O_l ∈ Unknown do
22             if ∃ l' : x.f = y ∈ R(O_l) : pts‾(O_l.f) ∩ CD ≠ ∅ then
23                 CD = CD ∪ {O_l}
24                 changed ← true
25     return CI ∪ (Unknown \ CD)
26 Procedure main()
27     Input: P                                              // Input program
28     Output: I                          // Set of Context-independent Objects
29     CI ← CD ← ∅
       /* Stage 1: Verifying Conch-P1, Conch-P2 and Conch-P3a               */
30     for O_l ∈ ℍ do
31         if ¬Is_Conch-P1_Valid(O_l) then
32             CI = CI ∪ {O_l}
33         else if ¬Is_Conch-P2_Valid(O_l) then
34             CI = CI ∪ {O_l}
35         else if Is_Conch-P3a_Valid(O_l) then
36             CD = CD ∪ {O_l}
   /* Stage 2: Verifying Conch-P3b                                          */
37 I = Check_Validity_Of_Conch-P3b(CI, CD)
38 return I
```

represent the sets of context-independent and context-dependent objects found so far, respectively, during the course of our analysis. According to Observation 3, Conch-P3 = Conch-P3a ∨ Conch-P3b. Our algorithm verifies Conch-P1 ∧ Conch-P2 ∧ Conch-P3 in two stages, with the first stage devoted to Conch-P1, Conch-P2, and Conch-P3a (lines 30–36) and the second stage to Conch-P3b (line 37).

$$\frac{l : x = \textbf{new } T}{O_l \xrightarrow{\text{new}} x \quad x \xrightarrow{\overline{\text{new}}} O_l}\ \text{[P-New]} \qquad \frac{x\ =\ y}{y \xrightarrow{\text{assign}} x \quad x \xrightarrow{\overline{\text{assign}}} y}\ \text{[P-Assign]}$$

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

Fig. 17. Constructing the PAG edges for a method with no parameters/returns/calls.

4.1 Verifying Conch-P1

According to [New] given in Figure 15, $\mathbb{H}$ contains the set of objects in P, with $O_l \in \mathbb{H}$ representing an object allocated in line l. Consider Algorithm 1 now. In lines 31 and 32, an object O_l is classified as being context-independent and thus inserted into CI if it does not satisfy Conch-P1, i.e., if Is_Conch-P1_Valid(O_l) (given in lines 1 and 2) fails to hold. Otherwise, we will proceed to verify Conch-P2 as described in Section 4.2 and Conch-P3 as described in Section 4.3.

4.2 Verifying Conch-P2

In lines 33 and 34, an object O_l is classified as being context-independent and thus inserted into CI if it does not satisfy Conch-P2, i.e., if Is_Conch-P2_Valid(O_l) (given in lines 3 and 4) fails to hold, or equivalently, if $O_l \notin$ *leakedObjects* holds, where *leakedObjects* contains the set of objects that may flow out of their allocating methods according to Observation 2. Otherwise, we will proceed to verify Conch-P3. To compute *leakedObjects* for P, we introduce an IFDS-based algorithm in Figure 22 that operates on its PAG [21] context-sensitively, based on the **Deterministic Finite Automaton** (**DFA**) given in Figure 20. Computing *leakedObjects* requires reasoning about object reachability in P. We describe our algorithm incrementally below.

Initially, we start with a parameterless method containing no calls. Its PAG can be built by applying the standard rules given in Figure 17 [21]. In the PAG, every non-call statement in the method is represented by two edges: a regular edge of the form $u \xrightarrow{\ell} v$ (where the label ℓ indicates the kind of statement represented (Table 1) and its inverse $v \xrightarrow{\overline{\ell}} u$, so that the same statement (edge) can be traversed forwards and backwards, respectively. Our object reachability analysis is designed to be field-insensitive, as reflected in [P-Load] and [P-Store], since we are only concerned with whether an object may leak out of its containing method rather than the specific field access path causing it to be leaked. Note that global loads `x = T.f` and global stores `T.f = y`, where `T.f` is a static field in class T, are ignored in the PAG, since static fields are always analyzed context-insensitively by $k$obj. Figure 19(a) gives a DFA for tracing approximately how an object O allocated in a method flows over the PAG. There are four states: H (starting at a heap object), F (moving forward in the PAG), B (moving backward in the PAG), and E (exiting from the allocating method). Due to the absence of parameters and returns, no object can flow out of a method, once it is allocated inside, as indicated by the lack of transitions into the final state E.

Let us explain the object reachability analysis, supported by this DFA (Figure 19(a)), on the PAG of a parameterless method. If the DFA starts with an object O under state H and transits to a node x under state F by following a sequence of PAG edges, then either O flows directly to x (via a new edge and possibly some assign edges) or O first flows into an access path $O'.f_1.\cdots.f_n = O$, where O', which is a locally allocated object, flows to x. If the DFA starts with an object O under state H and transits to a node y under state B, then either O is stored directly into an access path of y, i.e., $y.f_1.\cdots.f_n = O$, or O is first stored into an access path of some locally allocated object O' and then

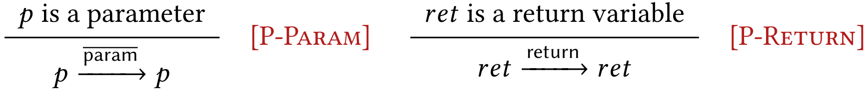
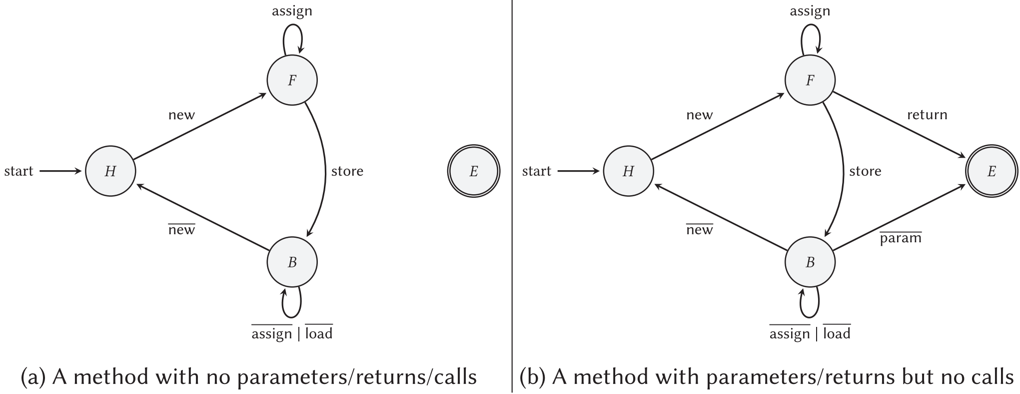


Fig. 18. Constructing the PAG edges for parameters and return variables.



(a) A method with no parameters/returns/calls (b) A method with parameters/returns but no calls

Fig. 19. Two intermediate DFAs for the DFA in Figure 20.

O' is stored into an access path of y, i.e., $y.f_1.\cdots.f_n = O'$. In this DFA, the load edges in the PAG are ignored as we track where O rather than its pointed-to objects flow to. In addition, the DFA also ignores the $\overline{\text{store}}$ edges in the PAG, as we assume that a method rarely contains a store and a load operating on the same field of an object (which is often accessed via its `getter` and `setter`). In the rare cases where this assumption fails to hold, *leakedObjects* may miss some objects that satisfy Conch-P2. As a result, this may cause Conch to mis-classify a context-dependent object as being context-independent, causing the underlying pointer analysis to lose some precision.

To support a method with parameters and a return variable but containing no calls, we add their self-loop edges using the rules given in Figure 18 and transform the DFA in Figure 19(a) into the one in Figure 19(b). Once an object allocated in a method flows to a parameter (suggested by $\overline{\text{param}}$) or the return variable (suggested by return) under state E, it is known to have leaked.

We now explain why we ignore the $\overline{\text{store}}$ edges in the PAG by using a small example in Figure 21. For the code given in Figure 21(a), object `B` is first stored into `A.f` (line 8) and then loaded into `b1` (line 9), implying that `b` and `b1` are aliases. Due to lines 10 and 11, object `C` will be leaked out of `foo()` via `b.g` and thus satisfies Conch-P2. However, *leakedObjects* will miss `C` since `a` $\xrightarrow{\overline{\text{store}}}$ `b` is ignored in the PAG for `foo()` depicted in Figure 21(b) and will thus not be handled by the DFA in Figure 19(b). It is easy to see that starting from the initial state H on `C`, the DFA cannot transit to the final state E. Note that we could have designed a more complex DFA that can handle the $\overline{\text{store}}$ edges, but doing so would cause many objects to be identified as being leaked spuriously. In practice, programmers will rarely write such code. As `b` and `b1` are aliases, a simpler solution that achieves the same functionality consists of deleting line 9 and replacing line 10 by `b.g = c`. For the thus modified code, *leakedObjects* will recognize `C` as being leaked as desired, since the DFA, when starting from the initial state H on `C`, will transit to the final state E by processing line 7 (`c = new C()`), line 10 (`b.g = c`), line 6 (`b = new B()`), and line 11 (`return b`) in that order.

Based on the two DFAs given in Figure 19, we obtain our final DFA given in Figure 20 for a whole program, where the three-dotted state transitions are added for handling call statements. For a given program, each method has its own PAG, but some summary edges can be added to its

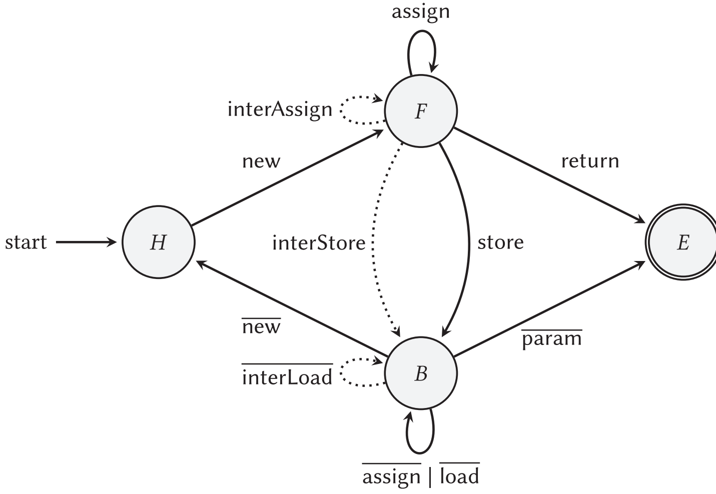
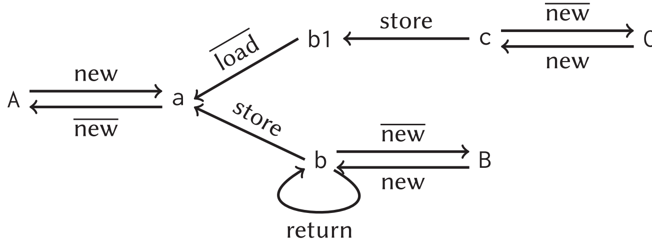


Fig. 20. The DFA used for verifying Conch-P2 for a whole program.

```
1 class A { B f; }
2 class B { C g; }
3 class C { }
4 Object foo() {
5     A a = new A();  // A
6     B b = new B();  // B
7     C c = new C();  // C
8     a.f = b;
9     B b1 = a.f;
10    b1.g = c;
11    return b;
12 }
```

(a) Code



(b) The PAG of `foo()`

Fig. 21. An example program for explaining why the $\overline{\text{store}}$ edges in its PAG are ignored.

PAG for its call sites during the analysis in order to capture the effects of inter-procedural value-flows across these call sites context-sensitively (as reviewed in Section 2.1), along the three-dotted state transitions. The PAG of the program consists of simply the PAGs for all its methods, with the call graph built according to the points-to information recorded in $\overline{pts}$. As discussed above, our object leak analysis is field-insensitive but context-sensitive operating on the PAG of the program.

Given a call statement $l : x = a_0.f(a_1, \ldots, a_r)$ contained in method m, let m' be a target method dispatched (with $p_i^{m'}$ being its ith parameter and $ret^{m'}$ being its return variable). Let n_1 and n_2 be two PAG nodes. We write $\langle n_1, S_1 \rangle \rightarrow \langle n_2, S_2 \rangle$ (as a path edge [40] in our setting) to indicate that node n_1 at state S_1 can reach node n_2 at state S_2. Let us write G_m to represent the PAG of method m. There are four cases considered when the callee method m' is analyzed:

- $\langle p_i^{m'}, F\rangle \rightarrow \langle p_j^{m'}, E\rangle$: $p_i^{m'}$ is saved into some access path of $p_j^{m'}$, i.e., $p_j^{m'}.f_1.\cdots.f_n = p_i^{m'}$, where each f_i represents some field. Then, we need to add a summary edge, $a_i \xrightarrow{\text{interStore}} a_j$ (i.e., $a_j.f = a_i$ for some field f), to G_m to propagate this reachability fact inter-procedurally.
- $\langle p_i^{m'}, F\rangle \rightarrow \langle ret^{m'}, E\rangle$: $p_i^{m'}$ is saved into some access path of a locally allocated object O in m', i.e., $O.f_1.\cdots.f_n = p_i^{m'}$, and then O flows out of m' via its return. Then, we need to add a summary edge, $a_i \xrightarrow{\text{interAssign}} x$, to G_m to reflect this reachability fact inter-procedurally.
- $\langle ret^{m'}, B\rangle \rightarrow \langle p_i^{m'}, E\rangle$: $ret^{m'}$ is loaded from some access path of $p_i^{m'}$, i.e., $ret^{m'} = p_i^{m'}.f_1.\cdots.f_n$. Then, we need to add a summary edge, $x \xrightarrow{\overline{\text{interLoad}}} a_i$ (i.e., $x = a_i.f$), to G_m to propagate this reachability fact inter-procedurally.
- $\langle O, H\rangle \rightarrow \langle ret^{m'}, E\rangle$: O, which is allocated in m', flows out of m' via its return statement. Then, we need to introduce a symbolic object Sym_l to abstract all the objects returned from the call site l by treating it effectively as an object allocation site that allocates Sym_l. We add two summary edges, which are actually the two PAG edges (one regular and one as its inverse) to represent the fact that x is assigned a new object Sym_l allocated at l.

Figure 22 gives our IFDS-based algorithm [40] for computing *leakedObjects*, operating on the PAG instead of the CFG of a program (for the first time). The rules in [SEEDS] inject three kinds of path edges, where the first one is for tracing which objects may leak out of their allocating methods and the other two are for finding summary edges. We have done one optimization to this algorithm. Traditionally [40], [SEEDS] will be applied to a method on-demand when it is found to be reachable from `main()` during the analysis. To improve parallelism in a parallel implementation, we inject the rules in [SEEDS] for all the methods in the call graph of the program at the beginning of our analysis in order to speed up the analysis at the cost of some redundant fact propagation. The rules in [PROPAGATE] perform the reachability analysis according to the DFA in Figure 20. Note that the three-dotted state transitions in the DFA are handled implicitly by the summary edges generated by [SUMMARY]. Finally, we collect the objects that can reach the final state, E, by using [COLLECT], indicating that these objects have leaked out of their allocating methods.

Let us illustrate the rules in Figure 22 by computing *leakedObjects* = {A1, A2} for our motivating example in Figure 4 (which reuses classes A and B from Figure 2). We focus on detecting A1 and A2 as two leaked objects, as illustrated in Figure 13, by deducing $\langle \texttt{A1}, H\rangle \rightarrow \langle \texttt{ret}^{\texttt{createA}}, E\rangle$ and $\langle \texttt{A2}, H\rangle \rightarrow \langle \texttt{this}^{\texttt{B}}, E\rangle$. All the other objects are not leaked, as explained in Section 2.3.3.

We perform our IFDS-based object reachability analysis on all the methods simultaneously. Let us consider `createA()` given in (Figure 2) first. According to [SEEDS], we inject $\langle \texttt{A1}, H\rangle \rightarrow \langle \texttt{A1}, H\rangle$. Subsequently, we obtain $\langle \texttt{A1}, H\rangle \rightarrow \langle \texttt{a1}, F\rangle$, $\langle \texttt{A1}, H\rangle \rightarrow \langle \texttt{ret}^{\texttt{createA}}, F\rangle$, and $\langle \texttt{A1}, H\rangle \rightarrow \langle \texttt{ret}^{\texttt{createA}}, E\rangle$ successively by applying the first, second, and seventh rules in [PROPAGATE] to process "`a1 = new A() // A1`" (line 20), $\texttt{ret}^{\texttt{createA}}$ `= a1`, and return $\texttt{ret}^{\texttt{createA}}$ in that order. This enables us to conclude immediately that A1 is leaked (from its allocating method) due to the second rule in [COLLECT]. According to [SEEDS] again, we also inject $\langle \texttt{p}, F\rangle \rightarrow \langle \texttt{p}, F\rangle$. Subsequently, we obtain $\langle \texttt{p}, F\rangle \rightarrow \langle \texttt{a1}, B\rangle$ and $\langle \texttt{p}, F\rangle \rightarrow \langle \texttt{A1}, H\rangle$ successively by applying the third and fifth rules in [PROPAGATE] to process `a1.f = p` (line 21) and "`a1 = new A() // A1`" (line 20) in that order. By noting that $\langle \texttt{A1}, H\rangle \rightarrow \langle \texttt{ret}^{\texttt{createA}}, E\rangle$, we can apply the sixth rule in [PROPAGATE] to obtain $\langle \texttt{p}, F\rangle \rightarrow \langle \texttt{ret}^{\texttt{createA}}, E\rangle$, which enables a new summary edge, $\langle \texttt{a2}, F\rangle \rightarrow \langle \texttt{a3}, F\rangle$, to be added to the callsite `a3 = this.createA(a2)` in line 16 according to the second rule in [SUMMARY]. Meanwhile, the two summary edges, $\langle \texttt{a3}, B\rangle \rightarrow \langle Sym_{16}, H\rangle$ and $\langle Sym_{16}, H\rangle \rightarrow \langle \texttt{a3}, F\rangle$, are also added to this callsite, where Sym_{16} represents symbolically all the objects returned by `createA()`, according to the fourth rule in [SUMMARY]. If some field operations were made directly on a3 in our example, then they would be analyzed as the field accesses operating directly on Sym_{16}.

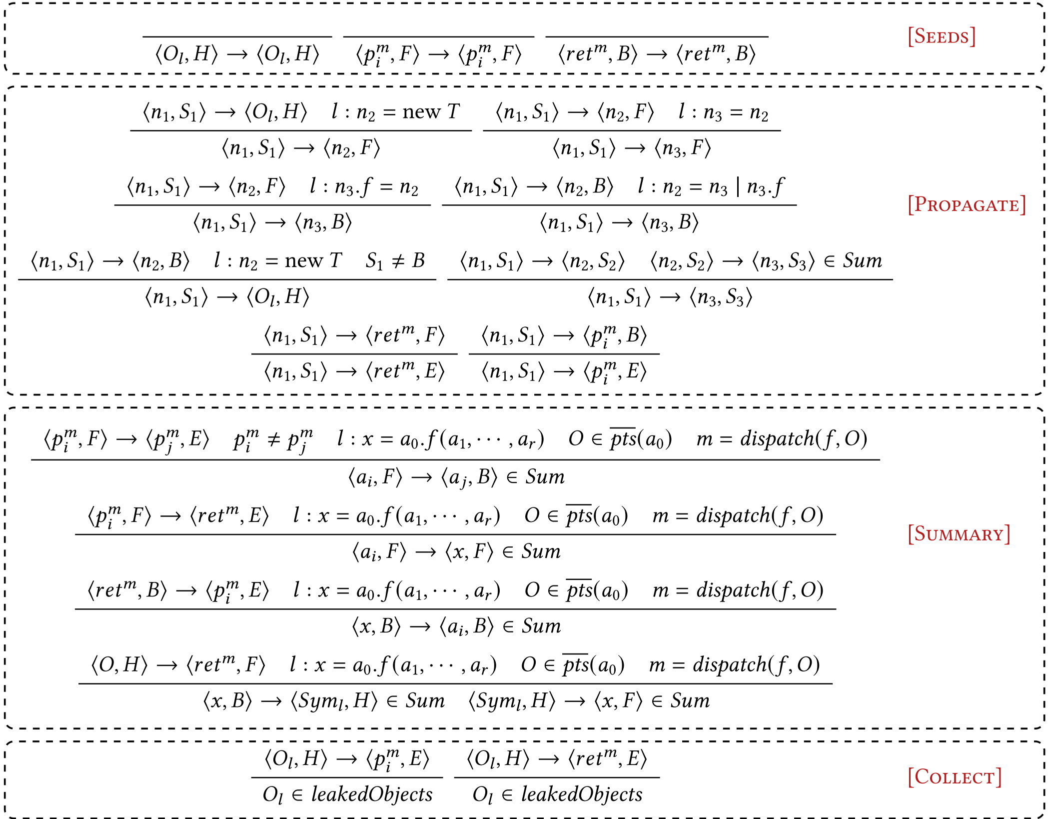


Fig. 22. Rules for computing *leakedObjects* to verify Conch-P2. To avoid duplicating rules unnecessarily, each rule that contains S_i (where $i \in \{1, 2, 3\}$) represents a set of rules in which $S_i \in \{H, F, B\}$, i.e., S_i can be either H or F or B. Sym_l is a symbolic object abstracting all the objects returned from call site l.

Let us now consider `B()` given in Figure 4. According to [Seeds], we inject $\langle \texttt{A2}, H\rangle \rightarrow \langle \texttt{A2}, H\rangle$. Due to "`a2 = new A() // A2`" (line 9), we find that $\langle \texttt{A2}, H\rangle \rightarrow \langle \texttt{a2}, F\rangle$ holds by applying the first rule in [Propagate]. Due to the presence of the summary edge $\langle \texttt{a2}, F\rangle \rightarrow \langle \texttt{a3}, F\rangle$ obtained earlier, we can establish $\langle \texttt{A2}, H\rangle \rightarrow \langle \texttt{a3}, F\rangle$ by applying the sixth rule in [Propagate]. Afterward, we obtain $\langle \texttt{A2}, H\rangle \rightarrow \langle \texttt{this}^{\texttt{B}}, B\rangle$ and $\langle \texttt{A2}, H\rangle \rightarrow \langle \texttt{this}^{\texttt{B}}, E\rangle$ successively by applying the third and eighth rules in [Propagate] to process `this.g = a3` and the parameter `this` in that order. Finally, we conclude that `A2` is also leaked due to the first rule in [Collect].

4.3 Verifying Conch-P3

According to Observation 3, we have Conch-P3 := Conch-P3a ∨ Conch-P3b. In lines 35 and 36 of Algorithm 1, we verify if an object O_l satisfies Conch-P3a by calling Is_Conch-P3a_Valid(O_l) (lines 5–14). If this is true, O_l is considered immediately as being context-dependent (and thus inserted into *CD*), since O_l has already satisfied both Conch-P1 and Conch-P2 at this point in our algorithm. Otherwise, we will proceed to verify Conch-P3b in line 37.

Let us examine Is_Conch-P3a_Valid(O_l). Given a store statement $x.f = y$, the key to verifying Conch-P3a (in determining the context-dependability of $O_l \in \overline{pts}(x)$) lies in the development of $depOnParam(y, m)$, which returns true if y is data-dependent on any parameter of the given method m. We introduce also an IFDS-based algorithm for computing $depOnParam$, in a similar

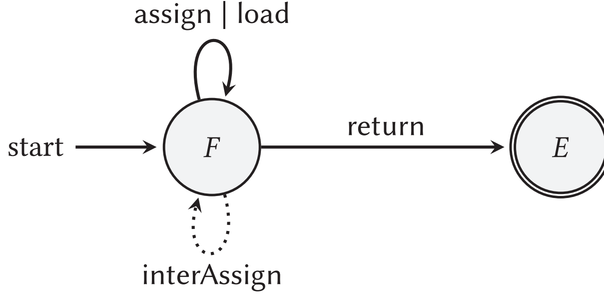


Fig. 23. The DFA used for computing *depOnParam*.

manner as how we have applied an IFDS-based algorithm for computing *leakedObjects*, by using a simpler DFA given in Figure 23 (than the DFA given in Figure 20).

This DFA has only two states, F and E, recognizing only four types of PAG edges, where interAssign is a summary edge introduced for supporting call statements exactly as in Figure 22. Given a call statement $l : x = a_0.f(a_1, \dots, a_r)$ in method m, let m' be a target method invoked. When $\langle p_i^{m'}, F\rangle \rightarrow \langle ret^{m'}, E\rangle$ happens, $ret^{m'}$ is recognized to be data-dependent on $p_i^{m'}$ (i.e., $ret^{m'} = p_i^{m'}.f_1.\cdots.f_n$). Thus, we add a summary edge, $a_i \xrightarrow{\text{interAssign}} x$, to the PAG of method m to propagate this reachability fact inter-procedurally from the callee m' to the caller m.

Our IFDS-based parameter dependency analysis algorithm, given in Figure 24, for computing *depOnParam* by proceeding forward from method parameters across the PAG of a program, is a simplified version of the one given in Figure 22. For [Seeds], only the parameters need to be injected (again simultaneously for all the methods in the call graph of the program to maximize parallelism in a parallel implementation). The rules for [Propagate] are similar. For [Summary], we use the summary edges added as discussed above. Finally, let $dps(v, m_v) = \{p_i^{m_v} \mid \langle p_i^{m_v}, F\rangle \rightarrow \langle v, F\rangle\}$ (with $\langle p_i^{m_v}, F\rangle \rightarrow \langle v, F\rangle$ indicating that v is data-dependent on the parameter $p_i^{m_v}$ of m_v), where v is a variable defined in its containing method m_v and $p_i^{m_v}$ is some (ith) parameter of m_v. Then $depOnParam(y, m)$ can be defined recursively (by taking care of chained constructors that are often used in practice, as illustrated by an example shortly) as follows:

$$depOnParam(y, m) = \begin{cases} dps(y, m_y) \neq \varnothing & \text{if } m = m_y \\ \bigvee_{p_i^{m_y} \in dps(y, m_y)} depOnParam(a_i, m) & \text{otherwise} \end{cases}, \tag{1}$$

where a_i is the corresponding argument of $p_i^{m_y}$ in a callsite to m_y.

Let us illustrate the rules given in Figure 24 by using two examples. For our motivating program given in Figure 4, it is straightforward to check that both A1 and A2 satisfy Conch-P3a, as shown in Figure 14. For A1 accessed in `a1.f = p` (line 15), where A1 ∈ $\overline{pts}$(a1), p is a parameter of m = `createA()`, established by ⟨p, F⟩ → ⟨p, F⟩, which is injected directly by [Seeds]. By Equation (1), $depOnParam(p, \texttt{createA}) = (dps(p, \texttt{createA}) \neq \varnothing) = (\{p\} \neq \varnothing) =$ true. Similarly, for A2 accessed in `this.f = o` (line 9), where A2 ∈ $\overline{pts}$(this), o is a parameter of m = `setF()`, established by ⟨o, F⟩ → ⟨o, F⟩, which is also injected directly by [Seeds]. By Equation (1), $depOnParam(o, \texttt{setF}) = (dps(o, \texttt{setF}) \neq \varnothing) = (\{o\} \neq \varnothing) =$ true. As A1 and A2 satisfy also Conch-P1 and Conch-P2, these two objects are found to be context-dependent.

Let us consider a more complex example in Figure 25 by verifying that the object B created in line 18 satisfies Conch-P3a, as illustrated in Figure 26. Given the store statement `this.g = r` in line 5 contained in the constructor `A()` (where $l' = 5$ in Algorithm 1), we find that the object B pointed to by `this.f` is allocated in the constructor `C()` (where m = `C` in Algorithm 1). To

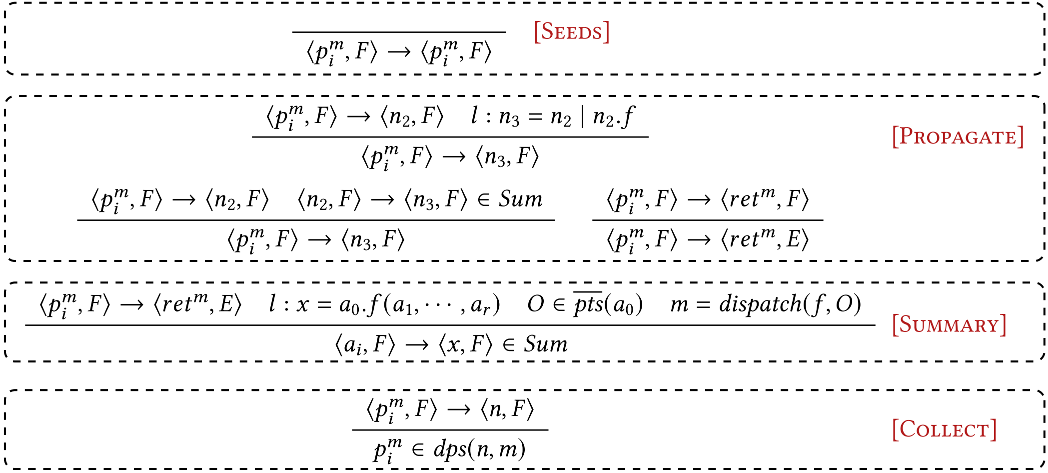


Fig. 24. Rules for computing *depOnParam* by means of computing *dps*.

```
1  class A {                          15 class C {
2    Object g;                        16   B f;
3    A (Object o) {                   17   C(Object t) {
4      Object r = this.id(o);         18     this.f = new B(t); // B
5      this.g = r;                    19   }
6    }                                20 }
7    Object id(Object x) {            21 void main() {
8      Object y = x;                  22   Object o1 = new Object(); // O1
9      return y;                      23   Object o2 = new Object(); // O2
10 }}                                 24   C c1 = new C(o1); // C1
11 class B extends A {                25   C c2 = new C(o2); // C2
12   B(Object p) {                    26   B t1 = c1.f, t2 = c2.f,
13     super(p);                      27   Object v1 = t1.g, v2 = t2.g;
14 }}                                 28 }
```

Fig. 25. An example for illustrating our IFDS-based analysis in verifying Conch-P3a.

show that B satisfies Conch-P3a, we need to show that $depOnParam(\mathtt{r}, \mathtt{C})$ = true. We are therefore required to show that (D1) $\mathtt{o} \in dps(\mathtt{r}, A)$ and (D2) $depOnParam(p, C)$ = true. To establish D2, we are required to show that (D3) $\mathtt{p} \in dps(\mathtt{p}, B)$ and (D4) $depOnParam(\mathtt{t}, C) = (dps(\mathtt{t}, C) \neq \emptyset) =$ true. As D3 and D4 hold trivially (due to $\langle \mathtt{p}, F\rangle \rightarrow \langle \mathtt{p}, F\rangle$ and $\langle \mathtt{t}, F\rangle \rightarrow \langle \mathtt{t}, F\rangle$ injected by [Seeds]), D2 is thus established. Below we show that D1 holds by applying the rules in Figure 24, as illustrated in Figure 26. As x is a parameter of `id()`, $\langle \mathtt{x}, F\rangle \rightarrow \langle \mathtt{x}, F\rangle$ holds according to [Seeds]. Subsequently, we obtain $\langle \mathtt{x}, F\rangle \rightarrow \langle \mathtt{y}, F\rangle$ and $\langle \mathtt{x}, F\rangle \rightarrow \langle \mathtt{y}, E\rangle$ successively by applying the first and the third rules in [Propagate] to process `y = x` (line 8) and `return y` (line 9) in that order. By applying [Summary], we add a summary edge, $\langle \mathtt{o}, F\rangle \rightarrow \langle \mathtt{r}, F\rangle$ to the callsite in line 4. As o is a parameter of `A()`, $\langle \mathtt{o}, F\rangle \rightarrow \langle \mathtt{o}, F\rangle$ holds according to [Seeds]. This enables us to obtain $\langle \mathtt{o}, F\rangle \rightarrow \langle \mathtt{r}, F\rangle$ by applying the second rule in [Propagate]. According to [Collect], we can now conclude that $\mathtt{o} \in dps(\mathtt{r}, A)$ and thus establish D1. Note that the object B is actually context-dependent as it also satisfies Conch-P1 (due to "`this.g = r`" (line 5) and "`v1 = t1.g/v2 = t2.g`" (line 27)) and Conch-P2 (due to "`this.f = new B(t)`" (line 18), causing B to leak out of `C()`).

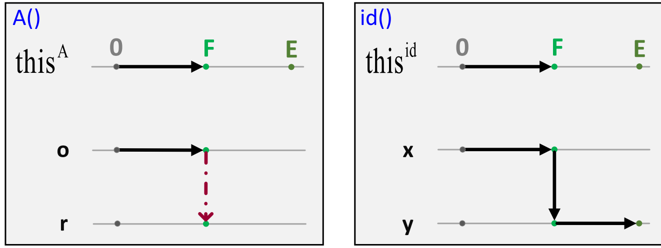


Fig. 26. An IFDS-based parameter dependency analysis performed by Conch on the PAG of the program given in Figure 25 to verify Conch-P3a for the object B created in line 18.

In the second stage of our Conch approach (line 37), Conch-P3b can be verified straightforwardly by calling Check_Validity_Of_Conch-P3b(*CI*, *CD*) (lines 15–25). At this point, *CI* and *CD* contain the sets of context-independent and context-dependent objects found so far during the course of our analysis. Let $O \in Unknown$, where $Unknown = \mathbb{H} \setminus (CI \cup CD)$ (line 16). Then O is found to be context-dependent if one of its fields can point to any context-dependent object (found so far) transitively and context-independent otherwise (lines 17–25).

4.4 Soundness and Precision

To place this research in a formal setting, we discuss the soundness and precision of Conch in two separate theorems below, even though their proofs are reasonably straightforward.

Theorem 1 (Soundness). *Let Conch be used to debloat contexts for an object-sensitive pointer analysis algorithm PTA with respect to a program. If PTA is sound (by never under-approximating the points-to information), then its context-debloated version, PTA_{Conch}, is also sound for the program.*

Proof. For a given pointer analysis algorithm *PTA*, its context-debloated version PTA_{Conch} for a program is obtained by applying Conch to *PTA* as shown in Figure 6. If *PTA* is sound, then PTA_{Conch} must also be sound for the program. This is because PTA_{Conch} differs from *PTA* only in that PTA_{Conch} may analyze potentially more objects in the program context-insensitively than *PTA* does, causing PTA_{Conch} to be no more precise than *PTA*, and consequently, sound. □

Theorem 2 (Precision). *Let Conch be used to debloat contexts for an object-sensitive pointer analysis algorithm PTA with respect to a program. If $\neg(C1 \wedge C2)$ holds for every context-independent object in the program that satisfies $\neg$(Conch-P1 $\wedge$ Conch-P2 $\wedge$ Conch-P3) and is thus included in CI by Conch, then its context-debloated version achieves the same precision as PTA for the program.*

Proof. As discussed in Section 2.3.2, C1 $\wedge$ C2 is a necessary condition for *k*obj to analyze an object O in a program context-sensitively without losing any precision [32]. Thus, If $\neg$(C1 $\wedge$ C2) holds for every context-independent object in *CI* found in the program by Conch (where $\neg$(Conch-P1 $\wedge$ Conch-P2 $\wedge$ Conch-P3) holds according to Algorithm 1), then its context-debloated version must achieve exactly the same precision as *PTA* for the program. □

As motivated in Section 2.3.3, Conch-P1 $\wedge$ Conch-P2 $\wedge$ Conch-P3 is developed to approximate C1$\wedge$C2 reasonably accurately according to Observation 1–Observation 3, as illustrated in Figure 8. The effectiveness of Conch for handling real-world programs will be demonstrated in Section 5.

4.5 Time Complexity

The worst-case time complexity of Conch in applying Algorithm 1 to separate the context-dependent and context-independent objects in a program P is linear in terms of the number of

its statements (or equivalently, the number of its PAG edges). Algorithm 1 proceeds in two stages, with the first stage devoted to verifying Conch-P1, Conch-P2 and Conch-P3a (lines 30–36) and the second stage devoted to verifying Conch-P3b (line 37).

The first stage is dominated by the process for verifying Conch-P2, i.e., computing *leakedObjects* for P according to an IFDS-based algorithm in Figure 22, since Conch-P1 can be verified in $O(|\mathbb{H}|)$ and the IFDS-based algorithm for verifying Conch-P3a has the same time complexity in the worst case. As reviewed in Section 2.1, Conch-P2 can be checked in $O(|E^*| \cdot |D|^3)$ [40], where $|E^*|$ is the number of PAG edges in P, which are constructed linearly to the number of statements in P according to Figures 17 and 18, and $D = 4$ is the number of states of the DFA in Figure 20. Thus, the first stage of Algorithm 1 runs in $O(|\mathbb{L}|)$, where $\mathbb{L}$ is the set of statements in P.

The second stage of Algorithm 1 (for checking the condition in line 22 by calling Check_Validity_Of_Conch-P3b () in lines 15–25) can be implemented by leveraging a reverse topological order of the field points-to graph of P [53]) and thus be done in $O(|\mathbb{H}|)$, where $\mathbb{H}$ is the set of heap objects in P. As $|\mathbb{H}| < |\mathbb{L}|$, the time complexity of Algorithm 1 is $O(|\mathbb{L}|)$.

5 EVALUATION

We demonstrate the effectiveness of our Conch approach by addressing five research questions:

- **RQ1.** Is Conch precise (measured in terms of the precision loss of an object-sensitive pointer analysis due to context debloating) and efficient (measured in terms of the time taken for separating the context-dependent from context-independent objects in a program)?
- **RQ2.** Can Conch speed up existing object-sensitive analysis algorithms significantly?
- **RQ3.** How effective are Observations 1–3 individually in helping Conch separate the context-dependent objects from context-independent objects in a program?
- **RQ4.** Can Conch be extended easily to make precision-efficiency tradeoffs?
- **RQ5.** Is the effectiveness of Conch sensitive to benchmark selections?

Implementation. We have implemented Conch in Soot [55], a program analysis and optimization framework for Java, on top of its context-insensitive Andersen's pointer analysis, Spark [21] (for computing $\overline{pts}$). Conch is implemented in Java and has been open-sourced at http://www.cse.unsw.edu.au/~corg/tools/conch. As described in Section 2, Conch aims to boost the performance of all object-sensitive pointer analysis algorithms at no or little loss of precision. We report and analyze our results by applying Conch to debloat three representative baseline algorithms, $k$obj (an object-sensitive version of Spark), Eagle [30, 32] (which is the only precision-preserving algorithm for accelerating $k$obj by supporting selective context-sensitivity) and Zipper [24] (the latest version b83b038, which can deliver the arguably best speedups for $k$obj among all the non-precision-preserving algorithms proposed for supporting selective context-sensitivity [11, 13, 18, 24]).

Experimental Setting. $k$obj is a standard implementation of Spark in Soot [10]. For Zipper [24, 26], a pre-analysis developed initially in Soot but released later in Doop [43], we will use it here to accelerate $k$obj in Soot. To reproduce its performance fairly, we have used an analysis setting that is as close as possible to the one used by the Zipper authors [24, 26] in several major aspects. First, we perform an exception analysis on the fly with $k$obj as in Doop by handling exceptions along the so-called exception-catch links [5]. Second, we use the declared type of an array element instead of `java.lang.Object` to filter type-incompatible points-to objects. Third, we handle native code by using the summaries provided in Soot. Fourth, we analyze a static method m by using the contexts of m's closest callers that are instance methods (on the call stack) and resolve Java reflection by using the reflection log generated by Tamiflex [4] as is often done in the pointer analysis literature [24, 44, 53]. Finally, objects that are instantiated from `StringBuilder`

and StringBuffer, as well as Throwable (including its subtypes), are distinguished per dynamic type and then analyzed context-insensitively as is done in Doop [6] and WALA [7]. Note that turning these manual heuristics on aims to challenge Conch to demonstrate its performance benefits over faster baselines by debloating the contexts for the faster baselines (already debloated manually). Otherwise, the speedups achieved by context debloating will be even more impressive.

To evaluate RQ1–RQ4, we have selected a set of 13 popular Java programs, including 10 benchmarks from DaCapo (consisting of 11 benchmarks released in 2006) [3], and 3 Java applications (checkstyle, JPC, and findbugs). These are the standard Java programs that are frequently used for evaluating pointer analysis algorithms [24, 44, 53]. For each program, its default reflection log that comes with it is used for resolving reflective calls. We have excluded jython from this DaCapo benchmark suite since its context sensitive analyses do not scale due to overly conservative handling of Java reflection [52]. The Java library used is jre1.6.0_45.

To evaluate RQ5, we have selected nine benchmarks from a more recent version of DaCapo (DaCapo-9.12) downloaded from Doop Benchmarks (consisting of also 11 benchmarks in total), representing relatively larger and more complex Java programs used in practice. We have excluded eclipse and jython since neither can be analyzed to completion by any of the three baseline pointer analysis algorithms under our time budget even when $k = 2$. For each benchmark, the reflection log used is its default version (with the suffix -tamiflex-default.log). For these nine new DaCapo benchmarks, a relatively larger and more complex Java library, jre1.8.0_121_debug, is used.

We have conducted our experiments on an Intel(R) Xeon(R) CPU E5-1660 3.2 GHz machine with 256 GB of RAM. The time budget used for running each pointer analysis on a program is set as 12 hours. For each pointer analysis, the analysis time of a program is an average of three runs.

5.1 RQ1: Is Conch Precise and Efficient?

Overall, Conch can speed up all the three baseline algorithms together substantially for the set of 13 programs evaluated (achieving 2.9× on average with a maximum of 15.9×), at no loss of precision for 11 programs and only a negligible loss of precision (less than 0.1%) for the remaining two programs. In addition, Conch can also improve the scalability of the three baseline algorithms substantially by enabling them to analyze 6 more programs under 11 configurations to completion than before (under a time budget of 12 hours), in a total of 8.3 hours.

Given Base (a baseline pointer analysis) and Base+D (Base with its contexts debloated by Conch), we measure the precision of Conch in terms of precision loss incurred with respect to a given metric (Metric) when both Base and Base+D are applied to analyze the same program:

$$\frac{\text{Metric(Base+D)} - \text{Metric(Base)}}{\text{Metric(Base)}}, \tag{2}$$

where Metric(Base) and Metric(Base+D) are the metric numbers obtained by Base and Base+D, respectively. We use a set of four common metrics for measuring the precision of a context-sensitive pointer analysis [24, 32, 44, 52]: (1) *#fail-cast*: the number of type casts that may fail, (2) *#call-edges*: the number of call graph edges discovered, (3) *#poly-calls*: the number of polymorphic calls discovered, and (4) *#reach-mtds*: the number of reachable methods.

Table 2 gives our main results. For $k$obj, E$k$obj (Z$k$obj) denotes the version of $k$obj with selective context-sensitivity provided by Eagle (Zipper). All these baselines (where $k \in \{2, 3\}$) and their debloated counterparts are compared by using the 13 Java programs considered.

Conch is very precise in terms of supporting context debloating while losing negligible precision. Our approach preserves the precision of all the baselines for 11 programs consisting of the 10 DaCapo benchmarks and findbugs. For checkstyle and JPC, our approach suffers from an

Table 2. Main Results

Prog	Metrics	Classic *k*OBJ				Eagle-guided *k*OBJ				Zipper-guided *k*OBJ			
		2OBJ	2OBJ+D	3OBJ	3OBJ+D	E2OBJ	E2OBJ+D	E3OBJ	E3OBJ+D	Z2OBJ	Z2OBJ+D	Z3OBJ	Z3OBJ+D
antlr	Time (s)	45.4	13.9(**3.3x**)	1049.3	185.4(**5.7x**)	28.5	12.3(**2.3x**)	732.7	167.8(**4.4x**)	20.8	7.8(**2.7x**)	337.9	32.1(**10.5x**)
	#fail-cast	509	509	449	449	509	509	449	449	559	559	507	507
	#call-edges	51176	51176	51149	51149	51176	51176	51149	51149	51394	51394	51367	51367
	#poly-calls	1622	1622	1615	1615	1622	1622	1615	1615	1643	1643	1636	1636
	#reach-mtds	7804	7804	7803	7803	7804	7804	7803	7803	7842	7842	7841	7841
bloat	Time (s)	743.8	359.5(**2.1x**)	**>12h**	4093.7	528.9	233.7(**2.3x**)	OoM	2059.8	532.4	279.2(**1.9x**)	OoM	2771.2
	#fail-cast	1314	1314	-	1221	1314	1314	-	1221	1368	1368	-	1279
	#call-edges	56699	56699	-	56464	56699	56699	-	56464	57192	57192	-	57036
	#poly-calls	1695	1695	-	1675	1695	1695	-	1675	1732	1732	-	1716
	#reach-mtds	9021	9021	-	9005	9021	9021	-	9005	9093	9093	-	9085
chart	Time (s)	253.0	85.5(**3.0x**)	OoM	4215.9	128.4	67.1(**1.9x**)	OoM	2723.3	34.6	20.3(**1.7x**)	573.6	178.3(**3.2x**)
	#fail-cast	1348	1348	-	1241	1348	1348	-	1241	1418	1418	1323	1323
	#call-edges	72457	72457	-	72023	72457	72457	-	72023	73123	73123	72738	72738
	#poly-calls	2032	2032	-	2008	2032	2032	-	2008	2060	2060	2040	2040
	#reach-mtds	15143	15143	-	15113	15143	15143	-	15113	15269	15269	15247	15247
eclipse	Time (s)	**>12h**	4097.5	OoM	OoM	4377.0	3317.7(**1.3x**)	OoM	OoM	3118.1	2732.4(**1.1x**)	OoM	OoM
	#fail-cast	-	3215	-	-	3215	3215	-	-	3357	3357	-	-
	#call-edges	-	145763	-	-	145763	145763	-	-	146492	146492	-	-
	#poly-calls	-	8720	-	-	8720	8720	-	-	8737	8737	-	-
	#reach-mtds	-	19916	-	-	19916	19916	-	-	19985	19985	-	-
fop	Time (s)	18.6	10.5(**1.8x**)	572.3	177.8(**3.2x**)	12.5	7.3(**1.7x**)	519.0	144.6(**3.6x**)	9.2	5.1(**1.8x**)	113.3	28.1(**4.0x**)
	#fail-cast	395	395	336	336	395	395	336	336	444	444	400	400
	#call-edges	34120	34120	34100	34100	34120	34120	34100	34100	34343	34343	34323	34323
	#poly-calls	808	808	802	802	808	808	802	802	832	832	826	826
	#reach-mtds	7582	7582	7582	7582	7582	7582	7582	7582	7620	7620	7620	7620
hsqldb	Time (s)	21.9	10.3(**2.1x**)	825.2	260.4(**3.2x**)	13.4	6.8(**2.0x**)	751.3	241.3(**3.1x**)	9.3	5.4(**1.7x**)	143.7	37.9(**3.8x**)
	#fail-cast	406	406	354	354	406	406	354	354	457	457	413	413
	#call-edges	34767	34767	34740	34740	34767	34767	34740	34740	35002	35002	34975	34975
	#poly-calls	830	830	823	823	830	830	823	823	853	853	846	846
	#reach-mtds	6980	6980	6979	6979	6980	6980	6979	6979	7022	7022	7021	7021
luindex	Time (s)	19.4	9.1(**2.1x**)	555.3	197.7(**2.8x**)	12.6	6.2(**2.0x**)	504.5	178.3(**2.8x**)	9.4	4.9(**1.9x**)	129.5	31.0(**4.2x**)
	#fail-cast	394	394	340	340	394	394	340	340	448	448	398	398
	#call-edges	33495	33495	33468	33468	33495	33495	33468	33468	33728	33728	33701	33701
	#poly-calls	918	918	911	911	918	918	911	911	944	944	937	937
	#reach-mtds	7017	7017	7016	7016	7017	7017	7016	7016	7057	7057	7056	7056
lusearch	Time (s)	30.4	11.8(**2.6x**)	2225.7	252.1(**8.8x**)	22.1	8.2(**2.7x**)	2067.6	224.2(**9.2x**)	13.2	5.2(**2.5x**)	622.7	39.2(**15.9x**)
	#fail-cast	409	409	357	357	409	409	357	357	466	466	418	418
	#call-edges	36377	36377	36350	36350	36377	36377	36350	36350	36605	36605	36578	36578
	#poly-calls	1116	1116	1109	1109	1116	1116	1109	1109	1143	1143	1136	1136
	#reach-mtds	7669	7669	7668	7668	7669	7669	7668	7668	7707	7707	7706	7706
pmd	Time (s)	41.6	24.2(**1.7x**)	1236.1	257.0(**4.8x**)	29.7	17.7(**1.7x**)	1082.7	238.1(**4.5x**)	23.9	14.9(**1.6x**)	344.7	52.5(**6.6x**)
	#fail-cast	1432	1432	1367	1367	1432	1432	1367	1367	1514	1514	1461	1461
	#call-edges	59864	59864	59805	59805	59864	59864	59805	59805	60029	60029	59970	59970
	#poly-calls	2357	2357	2351	2351	2357	2357	2351	2351	2382	2382	2376	2376
	#reach-mtds	11841	11841	11841	11841	11841	11841	11841	11841	11880	11880	11880	11880
xalan	Time (s)	565.3	298.2(**1.9x**)	OoM	1632.1	308.8	196.6(**1.6x**)	OoM	1377.6	223.6	222.5(**1.0x**)	2487.7	1125.6(**2.2x**)
	#fail-cast	600	600	-	546	600	600	-	546	657	657	609	609
	#call-edges	46653	46653	-	46621	46653	46653	-	46621	46842	46842	46815	46815
	#poly-calls	1613	1613	-	1606	1613	1613	-	1606	1636	1636	1629	1629
	#reach-mtds	9659	9659	-	9657	9659	9659	-	9657	9701	9701	9700	9700
checkstyle	Time (s)	1014.6	349.1(**2.9x**)	**>12h**	OoM	608.7	309.3(**2.0x**)	OoM	OoM	404.4	226.5(**1.8x**)	OoM	4887.4
	#fail-cast	1130	1130	-	-	1130	1130	-	-	1206	1206	-	1117
	#call-edges	67039	67041	-	-	67039	67041	-	-	67854	67854	-	66892
	#poly-calls	2210	2210	-	-	2210	2210	-	-	2268	2268	-	2211
	#reach-mtds	12314	12314	-	-	12314	12314	-	-	12383	12383	-	12342
JPC	Time (s)	106.1	54.6(**1.9x**)	2163.3	240.6(**9.0x**)	71.7	42.3(**1.7x**)	1309.1	210.9(**6.2x**)	34.4	26.3(**1.3x**)	181.0	44.8(**4.0x**)
	#fail-cast	1356	1356	1206	1206	1356	1356	1206	1206	1431	1431	1278	1278
	#call-edges	80965	80978	79297	79310	80965	80978	79297	79310	81616	81629	79932	79945
	#poly-calls	4263	4264	4127	4128	4263	4264	4127	4128	4324	4325	4187	4188
	#reach-mtds	15508	15508	15161	15161	15508	15508	15161	15161	15582	15582	15232	15232
findbugs	Time (s)	1629.6	180.1(**9.0x**)	OoM	936.3	873.6	152.5(**5.7x**)	OoM	938.4	131.3	50.3(**2.6x**)	1890.0	186.8(**10.1x**)
	#fail-cast	2072	2072	-	1696	2072	2072	-	1696	2144	2144	1956	1956
	#call-edges	87915	87915	-	86993	87915	87915	-	86993	88567	88567	87741	87741
	#poly-calls	3655	3655	-	3621	3655	3655	-	3621	3670	3670	3643	3643
	#reach-mtds	16266	16266	-	16219	16266	16266	-	16219	16315	16315	16287	16287

In all metrics (except for speedups), smaller is better. Given a pointer analysis Base, Base+D is its debloated version by CONCH. OoM stands for "Out of Memory". For each precision metric, the result obtained by Base+D is highlighted in red if it is different from the result obtained by Base.

average precision loss of only less than 0.1% (across all the four metrics). Figure 27 gives a real-world code snippet abstracted from JDK that causes a context-debloated pointer analysis to lose precision in analyzing JPC. For the two invocations made in lines 22 and 23, where `firePropertyChange()` is invoked on the receiver object `P1` with its argument pointing to `A` in line 22 and the receiver object `P2` with its argument pointing to `B` in line 23, the method `firePropertyChange()` defined in lines 7–9 is first called in both cases. Then the method `firePropertyChange()` defined in line 10 is called for the first invocation (made in line 22) and the method `firePropertyChange()` defined in lines 13–16 is called for the second invocation (made in line 23). Therefore, the variable `o1` declared in line 14 can only point to the object `B` under 2OBJ. According to CONCH, however, the object `E` (created in line 8) is context-independent as it does not

```
1  class PropertyChangeEvent {
2    Object oldValue;
3    public PropertyChangeEvent(Object o) { this.oldValue = o; }
4    public Object getOldValue() { return this.oldValue; }
5  }
6  class PropertyChangeSupport {
7    void firePropertyChange(Object p) {
8      firePropertyChange(new PropertyChangeEvent(p));  // E
9    }
10   void firePropertyChange(PropertyChangeEvent e1) { }
11 }
12 class DesktopPropertyChangeSupport extends PropertyChangeSupport {
13   void firePropertyChange(PropertyChangeEvent e2) {
14     Object o1 = e2.getOldValue();
15     o1.equals(o1);
16 }}
17 void main() {
18     A a = new A();  // A
19     B b = new B();  // B
20     PropertyChangeSupport p1 = new PropertyChangeSupport();  // P1
21     PropertyChangeSupport p2 = new DesktopPropertyChangeSupport();  // P2
22     p1.firePropertyChange(a);
23     p2.firePropertyChange(b);
24 }
```

Fig. 27. An example abstracted from JPC (with the non-main code abstracted from JDK) to illustrate why a context-debloated pointer analysis loses precision, where the definitions of classes A and B are irrelevant.

satisfy Conch-P2. By analyzing E context-insensitively, 2obj+D will conflate A and B in oldValue in line 2, causing it to conclude imprecisely that o1 in line 14 may point to both A and B, and consequently, the call in line 15 is polymorphic. As for checkstyle, a context-debloated pointer analysis loses precision similarly, where a LineReader object created in method load(InputStream) of class java.util.Properties is misclassified as being context-independent by Conch since it fails to satisfy Conch-P2.

Conch is also highly efficient (as a pre-analysis). For each benchmark, Table 3 gives the pre-analysis times of Eagle [30, 32], Zipper [24], and Conch. To allow their efficiency to be compared, we have given separately the analysis time of Spark [21] (shared by all the three). Note that both Zipper and Conch are designed to be multi-threaded (with 8 threads used in our experiments). Conch is slightly faster than Zipper, Eagle, and Spark across all the 13 programs. On average, their analysis times are 2.2 seconds (Conch), 8.2 seconds (Zipper), 14.6 seconds (Eagle), and 11.0 seconds (Spark). Thus, Conch is efficient enough for supporting context debloating in practice.

5.2 RQ2: Can Conch Speed Up Baseline Analyses?

Table 2 also gives the analysis times of all the pointer analyses evaluated. Conch delivers significant speedups (geometric means) over all the baselines. For $k$obj, the speedups of 2obj+D over 2obj range from 1.7× (for pmd) to 9.0× (for findbugs) with an average of 2.5×. When $k = 3$, the

Table 3. Times Spent by All the Pre-analyses in Seconds

	antlr	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs
Spark	8.6	8.9	15.6	24.7	7.6	7.1	6.9	7.8	11.7	9.0	13.6	17.4	16.7
Eagle	8.2	9.1	26.0	76.7	6.5	9.5	6.3	6.9	21.2	8.7	23.6	25.9	31.0
Zipper	4.6	6.5	16.4	25.7	4.0	4.2	3.7	4.3	9.5	10.1	14.5	9.8	16.2
Conch	1.7	1.9	3.2	6.4	1.5	1.4	1.4	1.4	2.2	2.7	2.8	2.6	3.1

For each benchmark, the analysis time of Spark shared by Eagle, Zipper, and Conch is given separately, together with their respective analysis times.

speedups of 3obj+D over 3obj are more impressive, ranging from 2.8× (for `luindex`) to 9.0× (for `JPC`) with an average of 4.8×. For Eagle, the speedups of E2obj+D over E2obj range from 1.3× (for `eclipse`) to 5.7× (for `findbugs`) with an average of 2.1×. When $k = 3$, the speedups of E3obj+D over E3obj range from 2.8× (for `luindex`) to 9.2× (for `lusearch`) with an average of 4.5×. For Zipper, the speedups of Z2obj+D over Z2obj range from 1.0× (for `xalan`) to 2.7× (for `antlr`) with an average of 1.8×. When $k = 3$, the speedups of Z3obj+D over Z3obj are also more impressive, ranging from 2.2× (for `xalan`) to 15.9× (for `lusearch`) with an average of 5.4×.

In accelerating $k$obj, Eagle [32] is precision-preserving (so that E$k$obj always achieves the same precision as $k$obj) but Zipper [24] is not precision-preserving (so that Z$k$obj) is usually less precise than $k$obj). Note that for the four precision metrics considered in Table 2, *#fail-cast*, *#call-edges*, *#poly-calls*, and *#reach-mtds*, E$k$obj yields the same results as $k$obj and E$k$obj+D yields the same results as $k$obj+D in theory. For all the three baselines, $k$obj, E$k$obj and Z$k$obj, Conch can make them run significantly faster at no or little loss of precision (as motivated in Section 2.3.1).

These results suggest that the speedups delivered by Conch increase as k increases, implying that Conch can help a variety of object-sensitive pointer analysis algorithms improve their scalability as well. During our evaluation, we find that 2obj+D scales one more benchmark, i.e., `eclipse` than 2obj, 3obj+D can scale four more benchmarks (`bloat`, `chart`, `xalan`, and `findbugs`) than 3obj, E3obj scales four more benchmarks, `bloat`, `chart`, `xalan`, and `findbugs` than E3obj, and Z3obj+D can scale two more benchmarks (`bloat` and `checkstyle`) than Z3Obj. In general, an analysis may be unscalable due to running either out of memory ("OoM") or the time budget ("**>12h**").

Therefore, Conch can accelerate existing object-sensitive pointer analyses significantly with negligible loss in precision. These include not only $k$obj (the standard algorithm) but also its variants enabled by, e.g., Zipper [24] and Eagle [30, 32] (the two recent attempts on applying selective context-sensitivity to improve the performance of $k$obj).

Below we analyze in detail why context debloating can enable baseline analyses, $k$obj, E$k$obj, and Z$k$obj, to improve their efficiency and scalability (as reported in Table 2).

Figure 28 depicts the percentage distribution of context-dependent and context-independent objects classified by Conch. Conch has successfully identified a large percentage of context-independent objects in all the programs, ranging from 65.6% (in `eclipse`) to 78.7% (in `fop`) with an average of 72.6%. Therefore, a large number of precision-irrelevant contexts have been eliminated via context debloating (as motivated in Figure 5).

Table 4 compares the three baseline pointer analyses ($k$obj, E$k$obj, and Z$k$obj) and their context-debloated counterparts ($k$obj+D, E$k$obj+D, and Z$k$obj+D) in terms of the average number of contexts analyzed for a method, where $k \in \{2, 3\}$. The debloated pointer analyses have achieved a substantial reduction in terms of this important metric across all the programs, providing the reasons behind the improved efficiency and scalability via context debloating.

Finally, we can also understand the effectiveness of Conch from the substantial reduction it has achieved in the number of context-sensitive facts inferred. In Table 5, #cs-gpts, #cs-pts, and #cs-fpts

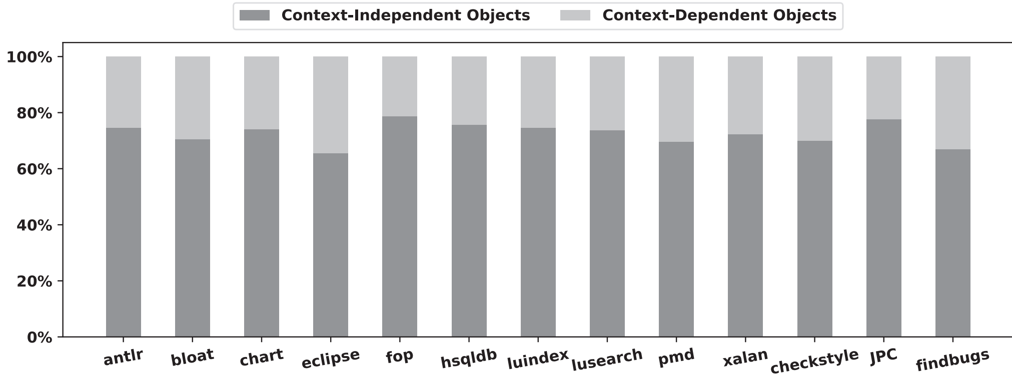


Fig. 28. Percentage distribution of the two types of objects.

Table 4. Average Number of Contexts Analyzed for a Method by the Three Baselines and their Context-Debloated Versions, kOBJ, kOBJ+D, EkOBJ, EkOBJ+D, ZkOBJ, and ZkOBJ+D, where $k \in \{2, 3\}$

	antlr	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs
2OBJ	27.1	30.3	36.5	-	15.6	19.4	17.1	20.0	17.3	50.8	66.4	24.7	37.9
2OBJ+D	13.1	18.3	20.8	31.1	9.2	10.5	9.9	10.3	10.3	31.6	41.4	16.0	18.8
E2OBJ	19.0	21.6	26.2	30.8	11.1	13.7	12.2	14.2	12.0	35.6	45.4	17.7	26.1
E2OBJ+D	9.5	12.7	15.4	20.7	6.7	7.6	7.2	7.4	7.3	22.5	28.9	11.8	13.2
Z2OBJ	8.1	14.3	6.9	14.9	4.9	5.8	5.6	6.0	6.1	15.3	18.7	7.4	9.6
Z2OBJ+D	4.8	8.7	5.2	12.3	3.6	4.1	4.0	4.0	4.4	11.8	14.4	6.1	7.3
3OBJ	99.8	-	-	-	53.0	65.0	58.1	91.5	53.5	-	-	87.2	-
3OBJ+D	24.6	39.0	78.9	-	19.1	22.5	21.3	22.2	18.7	61.6	-	26.1	29.5
E3OBJ	68.8	-	-	-	36.7	44.7	40.4	63.9	37.2	-	-	58.5	-
E3OBJ+D	17.5	26.8	57.0	-	13.5	15.8	15.0	15.7	13.1	43.5	-	18.5	20.5
Z3OBJ	26.5	-	21.7	-	14.3	16.6	16.7	23.5	17.6	60.7	-	14.5	25.7
Z3OBJ+D	8.2	17.3	12.5	-	6.4	7.4	7.1	7.3	7.1	22.6	57.5	7.7	10.8

represent the numbers of context-sensitive objects pointed by global variables (i.e., static fields), local variables and instance fields, respectively, and #cs-calls represent the number of context-sensitive call edges. In general, the speedups of a pointer analysis over a baseline come from a significant reduction in the number of context-sensitive facts computed by the baseline. For example, 2OBJ+D is significantly faster than 2OBJ for `findbugs` as its number of context-sensitive facts are significantly less than 2OBJ. Similarly, E3OBJ+D is also much faster than E3OBJ for `antlr` and Z3OBJ+D is also much faster than Z3OBJ for `lusearch`. However, the analysis time of a pointer analysis is known to be not linearly proportional to the number of context-sensitive facts computed [52]. Consider `xalan`. Z2OBJ+D has achieved a reduction of 16.2% over Z2OBJ in terms of the number of facts inferred but their analysis times are comparable.

5.3 RQ3: How Effective Are Observations 1–3 Individually?

We investigate the individual effectiveness of Observations 1–3, i.e., the effectiveness of each of their three induced conditions, Conch-P1, Conch-P2, and Conch-P3, in context debloating. We write $\text{Conch}^{\text{P}i}$ to represent a version of Conch in which only condition Conch-Pi is activated and the other two are ignored in Algorithm 1, where $1 \leqslant i \leqslant 3$. Thus, $\text{Conch}^{\text{P}i}$ will classify an object in a program as being context-dependent if Conch-Pi holds and context-independent otherwise.

Table 5. Context-Sensitive Facts

Prog	Metrics	Classic kOBJ				Eagle-guided kOBJ				Zipper-guided kOBJ			
		2OBJ	2OBJ+D	3OBJ	3OBJ+D	E2OBJ	E2OBJ+D	E3OBJ	E3OBJ+D	Z2OBJ	Z2OBJ+D	Z3OBJ	Z3OBJ+D
antlr	#cs-gpts	4.9K	2.1K	12.1K	2.5K	4.9K	2.1K	12.0K	2.5K	5.7K	2.3K	17.6K	2.7K
	#cs-pts	19.8M	3.6M	228.8M	32.1M	10.8M	2.2M	152.9M	28.4M	18.6M	3.3M	205.1M	10.4M
	#cs-fpts	0.6M	0.1M	13.6M	6.3M	0.6M	0.1M	13.6M	6.3M	0.6M	0.1M	13.7M	6.3M
	#cs-calls	5.4M	1.3M	87.5M	22.7M	4.0M	1.1M	67.8M	21.5M	1.9M	0.5M	22.7M	1.1M
	Total	25.8M	5.1M	329.8M	61.1M	15.4M	3.4M	234.3M	56.2M	21.1M	3.9M	241.6M	17.8M
bloat	#cs-gpts	3.1K	1.9K	-	2.3K	3.0K	1.9K	-	2.3K	3.9K	2.0K	-	2.4K
	#cs-pts	159.8M	68.0M	-	325.0M	109.7M	46.4M	-	229.3M	140.9M	53.6M	-	235.0M
	#cs-fpts	5.7M	4.6M	-	28.8M	5.7M	4.6M	-	28.8M	6.9M	4.6M	-	28.0M
	#cs-calls	47.1M	20.9M	-	112.0M	43.0M	19.6M	-	108.5M	38.2M	16.4M	-	74.0M
	Total	212.7M	93.5M	-	465.8M	158.3M	70.7M	-	366.5M	186.0M	74.5M	-	336.9M
chart	#cs-gpts	12.5K	6.9K	-	11.3K	12.4K	6.9K	-	11.3K	10.1K	5.5K	24.6K	6.9K
	#cs-pts	56.9M	20.8M	-	944.2M	33.8M	15.0M	-	653.7M	16.2M	6.9M	166.8M	55.1M
	#cs-fpts	1.1M	0.4M	-	19.6M	1.0M	0.4M	-	19.6M	0.7M	0.3M	21.7M	14.0M
	#cs-calls	20.0M	8.5M	-	332.8M	14.1M	6.3M	-	122.0M	2.5M	1.4M	26.7M	10.1M
	Total	78.0M	29.7M	-	1296.6M	48.9M	21.7M	-	795.4M	19.5M	8.6M	215.3M	79.1M
eclipse	#cs-gpts	-	7.8K	-	-	27.9K	7.8K	-	-	21.9K	8.0K	-	-
	#cs-pts	-	585.7M	-	-	562.6M	439.0M	-	-	601.9M	512.5M	-	-
	#cs-fpts	-	12.8M	-	-	17.2M	12.8M	-	-	16.7M	13.5M	-	-
	#cs-calls	-	345.2M	-	-	288.0M	242.6M	-	-	161.3M	147.3M	-	-
	Total	-	943.7M	-	-	867.9M	694.5M	-	-	779.9M	673.4M	-	-
fop	#cs-gpts	2.9K	1.8K	4.3K	2.0K	2.9K	1.8K	4.2K	2.0K	3.4K	1.9K	9.1K	2.2K
	#cs-pts	4.1M	1.2M	67.8M	27.5M	2.4M	0.9M	52.4M	25.0M	3.7M	1.1M	47.0M	7.9M
	#cs-fpts	0.2M	71.4K	8.0M	5.8M	0.2M	70.6K	8.0M	5.8M	0.2M	76.4K	8.2M	6.2M
	#cs-calls	1.3M	0.5M	31.0M	20.0M	1.0M	0.5M	26.9M	19.4M	0.5M	0.2M	5.1M	0.7M
	Total	5.6M	1.8M	106.7M	53.2M	3.6M	1.4M	87.3M	50.1M	4.4M	1.4M	60.4M	14.8M
hsqldb	#cs-gpts	3.0K	1.6K	4.4K	1.8K	3.0K	1.6K	4.2K	1.8K	3.7K	1.8K	10.0K	2.0K
	#cs-pts	5.6M	1.5M	90.5M	39.5M	3.3M	1.1M	71.5M	36.1M	4.7M	1.4M	55.9M	10.2M
	#cs-fpts	0.2M	75.3K	11.2M	8.4M	0.2M	74.9K	11.2M	8.4M	0.2M	85.4K	10.9M	8.4M
	#cs-calls	1.7M	0.6M	41.8M	29.0M	1.3M	0.5M	36.5M	28.2M	0.7M	0.3M	5.8M	0.8M
	Total	7.5M	2.2M	143.5M	76.9M	4.8M	1.7M	119.2M	72.7M	5.6M	1.8M	72.6M	19.4M
luindex	#cs-gpts	2.8K	1.6K	4.5K	2.0K	2.7K	1.6K	4.3K	2.0K	3.7K	1.8K	10.6K	2.2K
	#cs-pts	4.4M	1.4M	72.6M	31.4M	2.7M	1.0M	56.4M	28.6M	4.1M	1.2M	53.0M	8.5M
	#cs-fpts	0.2M	73.0K	9.0M	6.6M	0.2M	72.5K	9.0M	6.6M	0.2M	77.3K	9.0M	6.6M
	#cs-calls	1.4M	0.6M	34.1M	22.9M	1.0M	0.5M	30.0M	22.2M	0.6M	0.3M	5.6M	0.8M
	Total	6.0M	2.0M	115.6M	60.9M	3.9M	1.6M	95.4M	57.4M	4.9M	1.6M	67.7M	15.9M
lusearch	#cs-gpts	2.9K	1.6K	4.2K	1.8K	2.8K	1.6K	4.0K	1.8K	3.7K	1.8K	10.3K	2.1K
	#cs-pts	6.8M	1.6M	193.6M	37.8M	4.6M	1.2M	172.8M	34.6M	5.4M	1.4M	116.5M	10.1M
	#cs-fpts	0.2M	77.4K	11.0M	7.9M	0.2M	77.1K	11.0M	7.9M	0.2M	82.7K	10.3M	7.9M
	#cs-calls	3.1M	0.7M	149.3M	27.8M	2.5M	0.6M	129.5M	27.0M	1.1M	0.3M	41.8M	1.0M
	Total	10.1M	2.4M	353.9M	73.6M	7.4M	1.9M	313.3M	69.5M	6.7M	1.8M	168.6M	19.0M
pmd	#cs-gpts	3.4K	1.9K	5.1K	2.1K	3.4K	1.9K	4.9K	2.1K	5.3K	2.1K	21.3K	2.4K
	#cs-pts	12.7M	5.1M	142.9M	42.0M	8.1M	4.0M	106.7M	38.0M	14.9M	4.8M	171.1M	14.8M
	#cs-fpts	0.6M	0.3M	13.1M	8.4M	0.6M	0.3M	13.1M	8.4M	1.1M	0.4M	17.0M	9.0M
	#cs-calls	3.9M	2.0M	56.8M	29.1M	2.8M	1.5M	47.0M	27.8M	2.2M	1.0M	17.4M	1.9M
	Total	17.2M	7.3M	212.8M	79.6M	11.4M	5.8M	166.8M	74.3M	18.2M	6.2M	205.5M	25.7M
xalan	#cs-gpts	4.9K	2.9K	-	3.2K	4.9K	2.9K	-	3.2K	4.2K	2.8K	10.0K	3.2K
	#cs-pts	160.4M	49.0M	-	161.0M	84.3M	35.2M	-	138.3M	51.1M	41.5M	517.5M	123.6M
	#cs-fpts	6.3M	4.3M	-	15.7M	6.3M	4.3M	-	15.7M	5.4M	4.5M	33.1M	16.0M
	#cs-calls	49.6M	21.6M	-	103.4M	35.9M	17.2M	-	85.9M	14.6M	13.7M	86.0M	52.8M
	Total	216.3M	74.9M	-	280.2M	126.5M	56.7M	-	239.9M	71.2M	59.7M	636.6M	192.3M
checkstyle	#cs-gpts	7.7K	3.5K	-	-	7.7K	3.5K	-	-	10.8K	4.3K	-	5.2K
	#cs-pts	166.2M	44.7M	-	-	119.8M	29.0M	-	-	130.8M	38.3M	-	353.9M
	#cs-fpts	1.5M	0.4M	-	-	1.5M	0.4M	-	-	2.8M	0.6M	-	141.6M
	#cs-calls	86.5M	23.2M	-	-	39.9M	13.5M	-	-	24.1M	9.0M	-	79.8M
	Total	254.2M	68.3M	-	-	161.2M	42.9M	-	-	157.6M	47.9M	-	575.3M
JPC	#cs-gpts	7.3K	4.1K	21.3K	5.7K	7.2K	4.1K	20.8K	5.7K	7.0K	3.8K	16.4K	4.3K
	#cs-pts	27.8M	11.9M	606.3M	48.0M	18.7M	8.9M	324.3M	38.3M	13.2M	7.0M	67.3M	12.2M
	#cs-fpts	0.9M	0.3M	19.3M	7.2M	0.8M	0.3M	19.0M	7.2M	0.8M	0.3M	11.2M	6.7M
	#cs-calls	9.8M	5.5M	93.8M	28.8M	7.5M	4.6M	66.2M	26.3M	2.8M	2.0M	8.2M	2.0M
	Total	38.5M	17.7M	719.5M	84.1M	27.0M	13.8M	409.5M	71.9M	16.8M	9.4M	86.7M	20.8M
findbugs	#cs-gpts	34.1K	4.5K	-	6.0K	33.8K	4.5K	-	6.0K	11.0K	4.5K	43.8K	5.9K
	#cs-pts	358.2M	41.2M	-	126.9M	273.4M	36.3M	-	114.2M	58.6M	19.5M	553.2M	38.6M
	#cs-fpts	18.0M	1.0M	-	23.1M	17.9M	1.0M	-	23.1M	5.0M	1.0M	61.1M	23.9M
	#cs-calls	147.2M	13.3M	-	84.9M	72.0M	9.0M	-	80.0M	13.2M	5.8M	101.5M	5.9M
	Total	523.5M	55.5M	-	234.8M	363.4M	46.3M	-	217.2M	76.8M	26.2M	715.9M	68.4M

Figure 29 gives the number of context-independent objects identified by Conch and its three variants, Conch^{P1}, Conch^{P2}, and Conch^{P3}. By definition, the set of context-independent objects selected by Conch is the union of the sets of context-independent objects selected by its three variants individually. Thus, Conch has successfully identified more context-independent objects than each variant alone for all 13 programs. Let us examine these three variants themselves. Conch^{P1}

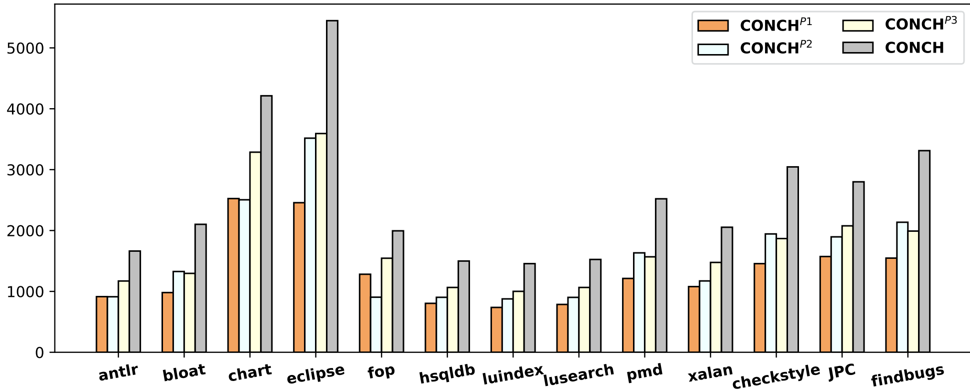


Fig. 29. Number of context-independent objects selected by Conch^{P1}, Conch^{P2}, Conch^{P3} and Conch.

is less effective than both Conch^{P2} and Conch^{P3} in identifying context-independent objects for all the 13 benchmarks except for `chart` and `fop`, where Conch^{P1} can find more context-independent objects than Conch^{P2} but still less than Conch^{P3}. Note that an object that contains no pointer fields will not satisfy Conch-P1. Conch^{P2} is more effective than Conch^{P3} in four benchmarks, i.e., `bloat`, `pmd`, `checkstyle`, and `findbugs`. For the other nine benchmarks, Conch^{P3} is more effective than Conch^{P2}. Note that Conch-P1, Conch-P2, and Conch-P3 are not mutually exclusive, as they can identify a common set of objects as context-independent in each benchmark.

We have selected Z2obj as the baseline to evaluate the individual contributions of Observations 1–3 toward the speedups achieved by context debloating for the 13 programs. Figure 30 depicts the speedups of Conch^{P1}, Conch^{P2}, Conch^{P3}, and Conch over this baseline. Note that, for each benchmark, the speedup achieved by Conch is not simply the sum of the speedups achieved by its three variants, since these variants may identify a common set of context-independent objects in the program (as mentioned above). Given this caveat, three conclusions can be made below. First, Conch achieves the best performance improvements than its three variants for all the 13 programs. Second, Conch^{P3} is more effective than both Conch^{P1} and Conch^{P2} in boosting the performance of Z2obj for all the 13 programs (except for `bloat` where Conch^{P3} underperforms Conch^{P2} substantially, `hsqldb` where Conch^{P3} underperforms Conch^{P1} visibly, and `xalan` where Conch^{P3} underperforms both Conch^{P1} and Conch^{P2} slightly), as it can generally identify more context-independent objects (as shown in Figure 29). Finally, Conch^{P1} is more effective than Conch^{P2} in the majority of the 13 programs despite the fact that Conch^{P1} usually finds fewer context-independent objects than Conch^{P2} (Figure 29). This is because many context-independent objects identified by Conch^{P1} are allocated in the JDK for encapsulating non-pointer primitive data and used universally in a range of real-world Java applications, where the JDK is used as a library.

5.4 RQ4: Can Conch be Extended Easily to Support Precision-Efficiency Tradeoffs?

As explained in Section 2.3, the key challenges faced in debloating an object-sensitive pointer analysis algorithm are to separate context-independent objects from context-dependent objects in a given program *accurately* and *efficiently* and enable the debloated pointer analysis algorithm to run *substantially faster at no or little loss of precision*. In developing Conch, our key insights are to identify a set of three conditions stated in Observations 1–3, Conch-P1, Conch-P2, and Conch-P3, that can separate context-dependent from context-independent objects reasonably accurately in real-world applications (Figure 8) and verify them efficiently by using an IFDS-based algorithm.

Given an object-sensitive pointer algorithm, Conch can be easily extended to achieve different precision-efficiency tradeoffs for its debloated counterparts. We demonstrate this by strengthening

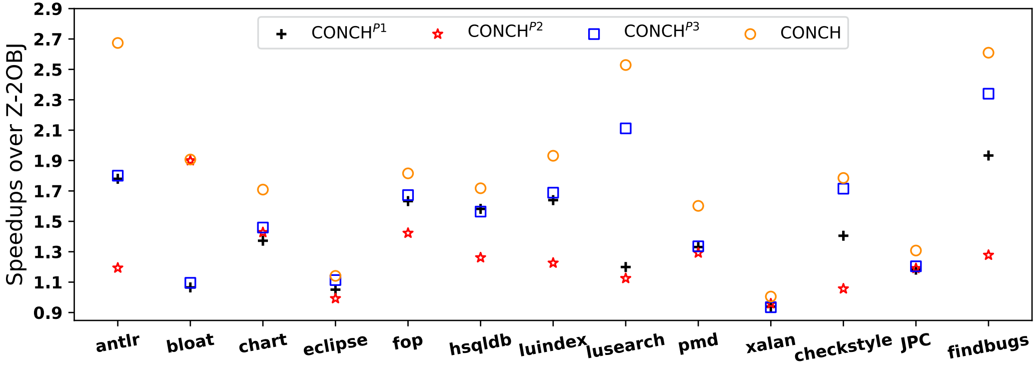


Fig. 30. The performance improvements by debloating Z2obj with Conch^{P1}, Conch^{P2}, Conch^{P3} and Conch.

```
1  class ProcessorTemplateElem {
2    void startElement(StylesheetHandler handler, ...) {
3      // The following line resolved from a reflective call to Class.newInstance()
4      ElemTemplateElement elem = new ElemForEach(); // E
5      handler.appendChild(elem);
6    }
7  }
```

Fig. 31. An example abstracted from xalan for illustrating why an object leaked out of its allocating method via a non-this parameter is often context-independent.

Conch-P2 to evolve Conch into a new version, Conch*, so that Conch* can deliver better performance improvements for some programs without losing any precision for the four metrics considered. According to Conch-P2 (i.e., its corresponding rules given in Figure 22), an object is regarded as being context-dependent if it can leak out of its allocating method via its this parameter or one of its other parameters (the first rule in [Collect]) or its return variable (the second rule in [Collect]). However, we observe that in real-world object-oriented programs (at least in our experiments), almost all the objects that leak out of their allocating methods via their non-this parameters are usually context-independent. This is possible because the concept of context dependability is used to capture the relation between an object and its allocator (pointed by this) while the non-this parameters are often not related to the allocator at all. Based on this observation, we can extend Conch into Conch* by strengthening the first rule in [Collect] in Figure 22 into the following version (while keeping the other rules unchanged):

$$\frac{\langle O_l, H\rangle \rightarrow \langle \texttt{this}^m, E\rangle}{O_l \in \mathit{leakedObjects}}$$

Figure 31 illustrates one representative scenario abstracted from xalan. The object E created in line 4 leaks out of its allocating method startElement() as it is stored into a field of the non-this parameter, handler, in the call to appendChild() in line 5. However, analyzing E context-insensitively does not affect the precision of $k$obj since handler points to a singleton object.

Table 6 gives the more context-independent objects identified by Conch* over Conch in both absolute and relative terms across the 13 programs. On average, Conch* has successfully identified 324.4 (4.8%) more context-independent objects than Conch.

Table 6. More Context-Independent (CI) Objects Identified by Conch* over Conch

More CI Objects	antlr	bloat	chart	eclipse	fop	hsqldb	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs
Abs.	298	396	322	368	298	297	297	302	331	335	318	322	333
Rel.	5.68%	6.65%	2.83%	2.55%	5.48%	6.26%	6.25%	6.02%	4.48%	5.28%	3.68%	3.71%	3.20%

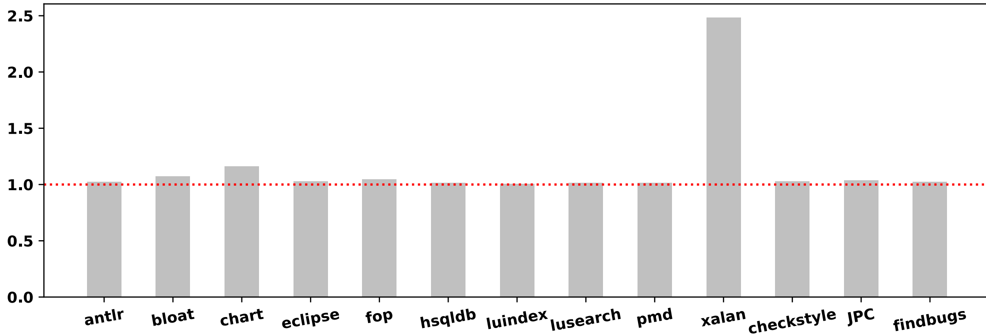


Fig. 32. The average performance improvement of Conch* over Conch.

Conch* achieves exactly the same precision as Conch (as given in Table 2) measured by all the four metrics for all the baselines across all the 13 programs. Therefore, we focus on comparing only the performance improvements achieved by Conch* and Conch. Figure 32 plots the speedups of Conch* over Conch for all the 13 programs (with the speedup for each program being computed as the average of the speedups achieved by Conch* over Conch for all the three baseline pointer analyses considered, i.e., $k$obj, E$k$obj, and Z$k$obj, given in Table 2). Conch* outperforms Conch substantially for `xalan` (at 2.48x) and `bloat` and `chart` visibly (at 1.06x and 1.16x, respectively), but performs no worse for the remaining 10 benchmarks. For all the 13 programs, the average performance improvement achieved by Conch* over Conch is 1.11x.

We have investigated why `xalan` benefits so significantly performance-wise from Conch*. In method `startElement()` of class `org.apache.xalan.processor.ProcessorTemplateElem`, a total of 119 objects, which are created via `Class::newInstance` and resolved by `Tamiflex` log, are identified as being context-dependent by Conch but context-independent by Conch*. As a result, the number of context-sensitive methods that are analyzed with these objects as their receivers has been significantly reduced (e.g., from 17,703 for 2obj debloated by Conch to 649 for 2obj debloated by Conch*). In general, the speedups achieved by Conch* over Conch are not proportional to the number of additional context-independent objects discovered by Conch* over Conch (Table 6). It is well-known that in $k$obj, different objects in a program affect its analysis time differently, depending on, for example, the number of methods invoked with these objects as receivers and where these objects appear in the OAG of the program (Figure 5). Given that Conch is open-sourced, Conch can be extended along the line suggested to make various precision-efficiency tradeoffs (by possibly trading precision for efficiency for some clients). In particular, how to extend Conch by accounting for the relative importance of different objects in determining the precision and efficient of a debloated pointer analysis algorithm will be a challenging future topic.

5.5 RQ5: Is the Effectiveness of Conch Sensitive to Benchmark Selections?

We have evaluated Conch further by using nine benchmarks from a more recent version of DaCapo (DaCapo-9.12) with a more recent version of the Java library (`jre1.8.0_121_debug`). Conch remains to be effective in boosting the performance of the three baseline pointer analysis

Table 7. Main Results for DaCapo-9.12

Prog	Metrics	Classic *k*obj				Eagle-guided *k*obj				Zipper-guided *k*obj			
		2OBJ	2OBJ+D	3OBJ	3OBJ+D	E2OBJ	E2OBJ+D	E3OBJ	E3OBJ+D	Z2OBJ	Z2OBJ+D	Z3OBJ	Z3OBJ+D
avrora	Time (s)	384.3	18.3(**21.0x**)	OoM	1097.4	367.0	15.7(**23.4x**)	OoM	1115.5	85.9	10.7(**8.0x**)	OoM	182.8
	#fail-cast	659	663	-	584	659	663	-	584	732	732	-	663
	#call-edges	53403	53403	-	53281	53403	53403	-	53281	53933	53933	-	53774
	#poly-calls	1236	1236	-	1208	1236	1236	-	1208	1277	1277	-	1262
	#reach-mtds	11822	11822	-	11811	11822	11822	-	11811	11873	11873	-	11859
batik	Time (s)	2422.9	624.5(**3.9x**)	OoM	OoM	1990.3	452.0(**4.4x**)	OoM	OoM	779.7	449.4(**1.7x**)	OoM	3013.3
	#fail-cast	2419	2421	-	-	2419	2421	-	-	2517	2517	-	2353
	#call-edges	122510	122510	-	-	122510	122510	-	-	123591	123591	-	121202
	#poly-calls	5658	5658	-	-	5658	5658	-	-	5709	5709	-	5650
	#reach-mtds	22766	22766	-	-	22766	22766	-	-	22843	22843	-	22793
h2	Time (s)	3680.6	556.2(**6.6x**)	OoM	OoM	2887.3	483.3(**6.0x**)	OoM	OoM	1112.2	296.6(**3.7x**)	OoM	OoM
	#fail-cast	1436	1448	-	-	1436	1448	-	-	1518	1518	-	-
	#call-edges	97021	97021	-	-	97021	97021	-	-	97910	97910	-	-
	#poly-calls	4025	4025	-	-	4025	4025	-	-	4080	4080	-	-
	#reach-mtds	15245	15245	-	-	15245	15245	-	-	15301	15301	-	-
luindex	Time (s)	391.2	16.5(**23.7x**)	OoM	912.8	366.7	15.1(**24.3x**)	OoM	893.0	87.5	10.0(**8.8x**)	OoM	142.6
	#fail-cast	553	557	-	459	553	557	-	459	636	636	-	557
	#call-edges	45314	45314	-	45124	45314	45314	-	45124	45858	45858	-	45669
	#poly-calls	1294	1294	-	1257	1294	1294	-	1257	1337	1337	-	1298
	#reach-mtds	9253	9253	-	9234	9253	9253	-	9234	9306	9306	-	9290
lusearch	Time (s)	391.8	16.5(**23.7x**)	OoM	941.5	373.9	14.6(**25.6x**)	OoM	966.9	87.8	9.5(**9.2x**)	OoM	147.6
	#fail-cast	499	505	-	425	499	505	-	425	577	577	-	509
	#call-edges	44056	44056	-	43966	44056	44056	-	43966	44639	44639	-	44521
	#poly-calls	1417	1417	-	1405	1417	1417	-	1405	1472	1472	-	1459
	#reach-mtds	9054	9054	-	9048	9054	9054	-	9048	9111	9111	-	9104
pmd	Time (s)	408.5	55.6(**7.3x**)	OoM	1488.3	399.0	45.7(**8.7x**)	OoM	1506.0	110.4	31.8(**3.5x**)	OoM	260.5
	#fail-cast	1055	1055	-	962	1055	1055	-	962	1140	1140	-	1056
	#call-edges	52680	52680	-	52562	52680	52680	-	52562	53523	53523	-	53334
	#poly-calls	1517	1517	-	1499	1517	1517	-	1499	1581	1581	-	1559
	#reach-mtds	11047	11047	-	11042	11047	11047	-	11042	11119	11119	-	11109
sunflow	Time (s)	1043.3	24.4(**42.8x**)	OoM	934.5	962.5	21.6(**44.6x**)	OoM	988.2	707.9	15.2(**46.6x**)	OoM	151.7
	#fail-cast	1377	1381	-	1285	1377	1381	-	1285	1507	1507	-	1410
	#call-edges	69450	69451	-	69115	69450	69451	-	69115	70145	70146	-	69772
	#poly-calls	2342	2342	-	2317	2342	2342	-	2317	2415	2415	-	2390
	#reach-mtds	15273	15273	-	15256	15273	15273	-	15256	15355	15355	-	15337
tradebeans	Time (s)	1197.7	22.1(**54.2x**)	OoM	3233.5	1111.1	18.5(**60.1x**)	OoM	3561.4	801.5	13.6(**58.9x**)	OoM	620.5
	#fail-cast	638	642	-	552	638	642	-	552	725	725	-	638
	#call-edges	48738	48738	-	48322	48738	48738	-	48322	49273	49273	-	48992
	#poly-calls	1396	1396	-	1373	1396	1396	-	1373	1442	1442	-	1417
	#reach-mtds	9920	9920	-	9893	9920	9920	-	9893	9963	9963	-	9952
xalan	Time (s)	683.7	252.8(**2.7x**)	OoM	2757.2	645.4	225.4(**2.9x**)	OoM	2705.4	1835.9	1758.4(**1.0x**)	OoM	3302.8
	#fail-cast	1162	1166	-	1085	1162	1166	-	1085	1290	1290	-	1220
	#call-edges	71396	71396	-	71249	71396	71396	-	71249	72206	72206	-	72015
	#poly-calls	3330	3330	-	3311	3330	3330	-	3311	3455	3455	-	3432
	#reach-mtds	13802	13802	-	13794	13802	13802	-	13794	13871	13871	-	13857

In all metrics (except for speedups), smaller is better. Given a pointer analysis Base, Base+D is its debloated version by Conch. OoM stands for "Out of Memory". For each precision metric, the result obtained by Base+D is highlighted in red if it is different from the result obtained by Base.

algorithms (*k*obj, E*k*obj, and Z*k*obj) considered while nearly preserving their precision. In fact, as these newer DaCapo benchmarks are relatively larger and more complex (causing the three baselines to suffer more severely from the context explosion problem (Figure 5)) than before, the speedups achieved by the debloated baselines, *k*obj+D, Z*k*obj+D, and Z*k*obj+D, are substantially more pronounced.

Below we discuss the results presented in Table 7.

5.5.1 Efficiency. Let us see how Conch improves the efficiency and scalability of the three baselines:

— $k = 2$**.** In this case, the three baselines, 2obj, E2obj, and Z2obj, and their respective debloated versions, 2obj+D, E2obj+D, and Z2obj+D, are all scalable (i.e., analyzed to completion under our time budget of 12 hours per benchmark). Figure 33 gives the speedups achieved by debloating each baseline across the nine benchmarks. Two remarks are in order here.

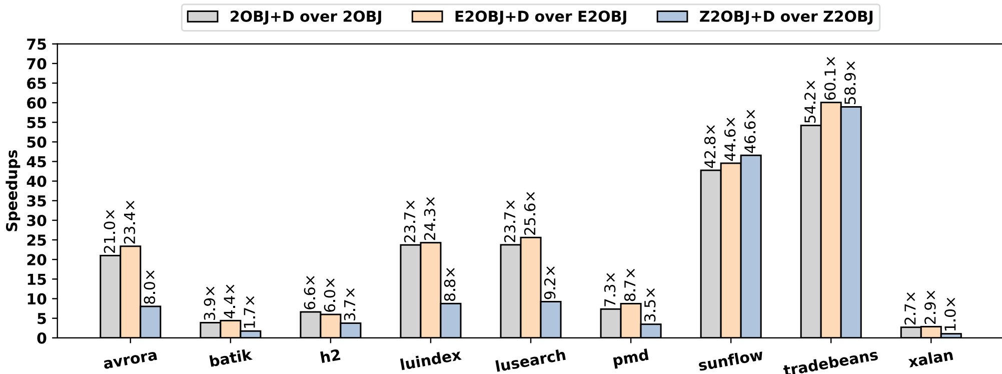


Fig. 33. The speedups of 2obj+D, E2obj+D, and Z2obj+D over 2obj, E2obj, and Z2obj, respectively.

First, Conch performs most impressively on `sunflow` (by accelerating each baseline by over 42×) and `tradebeans` (by accelerating each baseline by over 54×). In addition, Conch is also very effective on `avrora`, `luindex` and `lusearch`, but performs relatively more poorly on `batik`, `h2`, `pmd` and `xalan`. The average speedups achieved by 2obj+D, E2obj+D and Z2obj+D over 2obj, E2obj, and Z2obj are 13.4×, 14.3×, and 7.0×, respectively, which are substantially higher than those achieved for the older DaCapo benchmarks (Table 2). Second, for all the nine benchmarks except `sunflow` and `tradebeans`, the speedups achieved by Z2obj+D over Z2obj are noticeably lower than those achieved by 2obj+D over 2obj and E2obj+D over E2obj, since Zipper can identify more context-independent objects aggressively (at the expense of causing $k$obj to lose some precision) than Eagle (which identifies context-independent objects conservatively by preserving the precision of $k$obj) and $k$obj (which does nothing). However, for each baseline, its debloated baseline runs significantly faster due to additional context-independent objects identified successfully by Conch.

— $k = 3$**.** In this case, neither of the three baselines, 3obj, E3obj, and Z3obj is scalable (due to OoM). However, by debloating their contexts, their debloated versions, 3obj+D, E3obj+D, Z3obj+D, can now scalably analyze a common set of seven benchmarks, `avrora`, `luindex`, `lusearch`, `pmd`, `sunflow`, `tradebeans`, and `xalan`. In addition, Z3obj+D can also analyze `batik` scalably to completion. Overall, the total analysis times spent by 3obj+D, E3obj+D, Z3obj+D are 3.2 hours, 3.3 hours, and 2.2 hours, respectively.

5.5.2 Precision. For the three baselines, $k$obj, E$k$obj, and Z$k$obj, we only need to consider $k = 2$ as they are scalable when $k = 2$ but unscalable when $k = 3$ for all the nine benchmarks. Again, Conch causes the three debloated baselines (2obj+D, E2obj+D, and Z2obj+D) to lose no or little precision (less than 0.1% on average) with respect to their corresponding baselines (2obj, E2obj, and Z2obj).

Let us examine the three-pointer analysis algorithms considered in our evaluation. Z2obj+D exhibits a slight loss of precision in terms of *#call-edges* relative to 2obj in `sunflow` but otherwise achieves exactly the same precision as 2obj for all the four metrics, *#fail-cast*, *#call-edges*, *#poly-calls*, and *#reach-mtds*, across the nine benchmarks. Let us now move to the other two pointer analysis algorithms. For *#poly-calls* and *#reach-mtds*, 2obj+D and E2obj+D achieve the same precision as 2obj and E2obj across the nine benchmarks, respectively. For *#call-edges*, both 2obj+D and E2obj+D exhibit only a negligible loss of precision in `sunflow` relative to 2obj and E2obj, respectively. For *#fail-cast*, both 2obj+D and E2obj+D suffer from a slight loss of precision (an average of less than 0.1%) with respect to 2obj and E2obj, respectively (by reporting a few extra may-fail

casts as shown in Table 7). For both 2obj and E2obj, their worst-cases (in relative terms) happen in `lusearch` (where *#fail-cast* is 659 for both 2obj and E2obj but 663 for 2obj+D and E2obj+D).

We have also analyzed the reasons for the loss of precision caused by Conch. For the precision loss in *#call-edges* in `sunflow`, the code snippet given in Figure 27 is still the culprit. Due to a similar reason as illustrated in Figure 27, the precision loss in *#fail-cast* across the benchmarks is attributed to a `TimSort` object allocated in method `sort()` of class `java.util.TimSort` that is identified as being context-independent incorrectly (since it does not satisfy Conch-P2).

6 RELATED WORK

In this section, we mainly review the prior work that is the most closely related to improving the performance of whole-program pointer analysis for object-oriented programs.

There are several recent efforts on exploiting selective context-sensitivity to accelerate the performance of the standard object-sensitive pointer analysis (i.e., *k*obj) [24, 32]. Eagle [32] improves the efficiency of *k*obj while preserving its precision by conservatively reasoning about value flows via CFL reachability. Zipper [24], as a representative of non-precision-preserving approaches [11, 18, 24, 26], trades precision for efficiency by exploiting several value flow patterns. These techniques mitigate the context explosion problem of *k*obj by analyzing only a subset of methods in the program context-insensitively. In contrast, Conch represents a novel mitigation approach as it can debloat contexts for all the objects in the program, enabling existing algorithms to run significantly faster at only a negligible loss of precision.

There are other attempts on mitigating the context explosion problem in context-sensitive pointer analysis. By giving up precision-preserving guarantees offered by Eagle [32], Turner [13, 14] exploits object containment to provide better performance improvements in accelerating *k*obj. Conch can be used to boost its performance further. Mahjong [53] mitigates context explosion by merging equivalent heap abstractions at the expense of precision in alias relations. Conch is orthogonal to Mahjong and may boost its performance by debloating its contexts used.

Data-driven approaches [16–18] apply machine learning to obtain various heuristics for supporting selective context-sensitivity. Scaler [25] trades precision for scalability by selecting a suitable context-sensitivity variant for each method so that the amount of points-to information is bounded.

All the above pointer analysis techniques [13, 14, 16–18, 24, 25, 30, 53] share a common optimization pattern. They achieve their performance improvements by first exploiting a pre-analysis to select a set of precision-uncritical methods/variables/objects and then instructing the subsequent main pointer analysis to analyze the selected methods/variables/objects context-insensitively.

In [52], context transformations are introduced as an alternative context abstraction to context strings (as used in *k*obj), but the practical benefits are shown to be small.

Elsewhere [20, 33, 50, 51], efforts have been made to improve the precision of object-sensitive pointer analysis. This thread of research is orthogonal to ours considered here. It is worth noting that an insight discussed in [51] that is used for determining whether a method should be reanalyzed according to object sensitivity is quite similar to Conch-P2 despite that their solution seems to be more conservative than the IFDS-based object leak analysis proposed in this article.

Unlike whole-program analyses [6, 21–23, 35, 44, 58] considered in this article, demand-driven pointer analyses [41, 45, 46, 48, 49, 61] typically only compute the points-to information for program points that may affect a particular site of interest for specific clients.

Our object leak analysis (introduced in Section 4.2) is conceptually similar to escape analysis (for Java) [8, 59] but both are fundamentally different in terms of design objectives and hence runtime complexity. In terms of design objectives, our object leak analysis is performed to determine the context-dependability of an object by approximating its value flow based on CFL

reachability [30] while escape analysis is traditionally conducted to support register allocation, stack allocation, and synchronization optimization. As a result, our object leak analysis completely ignores static fields while escape analysis is sensitive to such fields. In addition, our object leak analysis is flow-insensitive and performed on a specifically customized PAG while escape analysis is usually flow-sensitive and performed on a control flow graph. In terms of runtime complexity, our object leak analysis is linear to the number of PAG edges so that it can be served as a lightweight pre-analysis for debloating the contexts for object-sensitive pointer analysis while the time complexities of escape analysis algorithms, which are formulated as data-flow analysis problems, are much higher [9]. As revealed in Table 3, Conch is extremely lightweight. Otherwise, the pre-analysis time incurred by Conch in a program may outweigh the performance improvement achieved from the underlying main analysis accelerated by Conch (based on our experience in designing pre-analyses).

7 CONCLUSION

Scalability is a major challenge in designing and developing precise object-sensitive pointer analysis techniques due to the combinatorial explosion of contexts in large object-oriented programs. In this article, we address this challenge by applying context debloating so that we can boost the performance of all object-sensitive pointer analysis algorithms with negligible loss in precision. Our key insight is to replace a set of two existing necessary conditions (whose verification is undecidable) with a set of three necessary conditions that can be linearly verified in terms of the number of PAG edges in a program for determining the context-dependability of the objects allocated in the program. Our evaluation shows that our new approach, Conch, can improve significantly the efficiency and scalability of not only $k$obj but also existing representative approaches to selective context-sensitivity that can already accelerate the performance of $k$obj.

We believe that the performance benefits of context debloating are not just limited to object-sensitive pointer analysis as demonstrated here. It would be interesting to explore how to apply context debloating to other flavors of pointer analysis such as call-site-sensitive pointer analysis [42] and context-transformation-based pointer analysis [52]. In addition, it would also be worthwhile investigating how to apply context debloating to other context-sensitive program analyses such as taint analysis [2] and data-dependence analysis [62] for improving their efficiency and scalability.

ACKNOWLEDGMENTS

Thanks to all the reviewers for their thoughtful comments and efforts towards improving our manuscript. This research is supported by an ARC Grant DP210102409.

REFERENCES

[1] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph. D. Dissertation. University of Cophenhagen.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[3] 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, 169–190.

[4] 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, 241–250.

[5] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception analysis and points-to analysis: Better together. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*. Association for Computing Machinery, New York, NY, 1–12.

[6] 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, 243–262.

[7] IBM T. J. Watson Research Center. 2022. WALA: T. J. Watson Libraries for Analysis. Retrieved April 27, 2022 from http://wala.sourceforge.net/.

[8] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. *Acm Sigplan Notices* 34, 10 (1999), 1–19.

[9] Alain Deutsch. 1997. On the complexity of escape analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. Association for Computing Machinery, New York, NY, 358–371. DOI: https://doi.org/10.1145/263699.263750

[10] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information flow analysis of Android applications in DroidSafe. In *NDSS*, Vol. 15. The Internet Society, 110.

[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, 13–18.

[12] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 267–279.

[13] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2021. Accelerating object-sensitive pointer analysis by exploiting object containment and reachability. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:31. DOI: https://doi.org/10.4230/LIPIcs.ECOOP.2021.16

[14] Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. 2022. Selecting context-sensitivity modularly for accelerating object-sensitive pointer analysis. In *Proceedings of the IEEE Transactions on Software Engineering*. IEEE, New York, NY, 1–1.

[15] Dongjie He, Jingbo Lu, and Jingling Xue. 2022. Qilin: A new framework for supporting fine-grained context-sensitivity in Java pointer analysis. In *Proceedings of the 36th European Conference on Object-Oriented Programming (ECOOP 2022)*, Vol. 222. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:29.

[16] 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.

[17] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[18] 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.

[19] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security (PLAS'06)*. Association for Computing Machinery, New York, NY, 27–36. DOI: https://doi.org/10.1145/1134744.1134751

[20] 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, 423–34.

[21] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In *Proceedings of the International Conference on Compiler Construction*. Springer, Berlin, 153–169.

[22] 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, 1110–1121. DOI: https://doi.org/10.1145/3540250.3549122

[23] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, 343–353.

[24] 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.

[25] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering. Association for Computing Machinery, New York, NY, 129–140.

[26] 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.

[27] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program tailoring: Slicing by sequential criteria. In *Proceedings of the 30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 15:1–15:27.

[28] Yin Liu and Ana Milanova. 2008. Static Analysis for inference of explicit information flow. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'08)*. Association for Computing Machinery, New York, NY, 50–56.

[29] V. Benjamin Livshits and Monica S. Lam. 2005. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium—Volume 14 (SSYM'05)*. USENIX Association, 18.

[30] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. 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.

[31] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Selective context-sensitivity for k-CFA with CFL-reachability. In *Proceedings of the International Static Analysis Symposium*. Springer International Publishing, Cham, 261–285.

[32] 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.

[33] Ana Milanova. 2007. Light context-sensitive points-to analysis for Java. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Association for Computing Machinery, New York, NY, 25–30.

[34] 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, 1–11.

[35] 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.

[36] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, 308–319.

[37] Erik M. Nystrom, Hong-Seok Kim, and Wen-Mei W. Hwu. 2004. Importance of heap specialization in pointer analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Association for Computing Machinery, New York, NY, 43–48.

[38] Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11-12 (1998), 701–726.

[39] Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 162–186.

[40] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. Association for Computing Machinery, New York, NY, 49–61.

[41] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*. Association for Computing Machinery, New York, NY, 264–274.

[42] Micha Sharir and Amir Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. S. S. Muchnick and N. D. Jones (Eds.), Prentice-Hall, Englewood Cliffs, NJ, Chapter 7, 189–234.

[43] Yannis Smaragdakis. 2021. Doop-Framework for Java Pointer and Taint Analysis (using P/Taint). Retrieved Jan 10, 2021 from https://bitbucket.org/yanniss/doop/.

[44] 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, 17–30.

[45] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java. In *Proceedings of the 30th European Conference on Object-Oriented Programming*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:26.

[46] 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, 387–400.

[47] 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, 112–122.

[48] 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*. Association for Computing Machinery, New York, NY, 59–76.

[49] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, 460–473.

[50] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *Proceedings of the International Static Analysis Symposium*. Springer, Berlin, 489–510.

[51] 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, 27–38. DOI: https://doi.org/10.1145/3377555.3377902

[52] 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, 263–277.

[53] 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, 278–291.

[54] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE'18)*. Association for Computing Machinery, New York, NY, 350–360. DOI: https://doi.org/10.1145/3180155.3180251

[55] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *Proceedings of the CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[56] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided Fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, New York, NY, 999–1010.

[57] Mark Weiser. 1984. Program slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 352–357. DOI: https://doi.org/10.1109/TSE.1984.5010248

[58] John Whaley and Monica S. Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, 131–144.

[59] John Whaley and Martin Rinard. 1999. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. Association for Computing Machinery, New York, NY, 187–206.

[60] Diyu Wu, Dongjie He, Shiping Chen, and Jingling Xue. 2020. Exposing android event-based races by selective branch instrumentation. In *Proceedings of the 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, New York, NY, 265–276.

[61] 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, 155–165.

[62] Qirun Zhang and Zhendong Su. 2017. Context-sensitive data-dependence analysis via linear conjunctive language reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. Association for Computing Machinery, New York, NY, 344–358.

Received 15 June 2022; revised 3 October 2022; accepted 5 December 2022