2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678880

Context Debloating for Object-Sensitive Pointer Analysis

Dongjie He, Jingbo Lu, and Jingling Xue
UNSW Sydney

***Abstract*—We introduce a new approach, CONCH, for debloating contexts for all the object-sensitive pointer analysis algorithms developed for object-oriented languages, where the calling contexts of a method are distinguished by its receiver objects. Our key insight is to approximate a recently proposed set of two necessary conditions for an object 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 statements in the program) that are almost always necessary for real-world object-oriented applications, based on three key observations regarding context-dependability for their objects used. To create a practical implementation, we introduce a new IFDS-based algorithm for reasoning about object reachability in a program. By debloating contexts for two representative object-sensitive pointer analyses applied to a set of 12 representative Java programs, CONCH can speed up the two baselines together substantially (3.1x on average with a maximum of 15.9x) and analyze 7 more programs scalably, but at only a negligible loss of precision (less than 0.1%).**

***Index Terms*—Pointer Analysis, Object Sensitivity, Debloating**

I. INTRODUCTION

Many software engineering tasks such as call graph construction [1], [2], program slicing [3], [4], program understanding [5], and bug detection [6]–[10] often require precise points-to/alias information. The quality of a pointer analysis directly determines the effectiveness and usefulness of the tools developed for accomplishing these tasks.

For object-oriented languages, object-sensitive pointer analysis, which distinguishes the (calling) contexts of a method by its receiver objects, is regarded as providing highly useful precision [11]–[15] and thus widely adopted in several pointer analysis frameworks for Java, such as SOOT [16], DOOP [17] and WALA [18]. Under k-object-sensitivity [19], [20], denoted kOBJ, a context used for analyzing a method m is represented by a sequence of k context elements (under k limiting), $[o_1, ..., o_k]$, 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 [11]. So o_i is an *allocator* of o_{i-1}.

Currently, kOBJ does not scale well for reasonably large programs when $k \geqslant 3$ and is often time-consuming when it is scalable [11]–[14]. As k increases, the number of contexts analyzed for a method often blows up exponentially without improving precision much. To alleviate this issue, several recent research efforts [15], [21]–[24] focus on selective context-sensitivity, which first conducts a pre-analysis to the program and then instructs kOBJ to apply context-sensitivity only to some of its methods selected. A number of attempts have been made, including client-specific machine learning techniques [21] (guided by improving the precision of a given client, e.g., may-fail-casting) and general-purpose techniques, such as user-supplied hints [23], [24], pattern matching [22], and CFL (Context-Free Language) reachability [15], [25], [26]. Despite some performance improvements obtained (at no or a noticeable loss of precision), these existing selective context-sensitive pointer analysis algorithms still suffer from an unreasonable explosion of contexts.

We introduce a new approach, CONCH, for debloating contexts for all object-sensitive pointer analysis algorithms, including kOBJ and its various incarnations for performing selective context-sensitivity, by boosting their performance significantly with negligible loss in precision. In object-oriented programs, we observe that a large number of objects that are allocated in a method are used independently of its calling contexts. Distinguishing these objects context-sensitively, as often done in the past, will serve to increase only the number of calling contexts analyzed for the methods invoked on these objects (as receivers) without any precision improvement.

Our key insight is to approximate a recently proposed set of two necessary conditions for an object to be context-sensitive, i.e., *context-dependent* [15], [25] (whose precise verification is undecidable [27]) with a set of three linearly verifiable necessary conditions (in terms of the number of statements in the program), based on three key observations regarding context-dependability for the objects used practically in real-world object-oriented programs. To create a practical implementation for CONCH, we have developed a new lightweight IFDS-based algorithm [28] for verifying these conditions (governing object reachability). By allowing only context-dependent objects to be handled context-sensitively, CONCH can significantly limit the explosive growth of the number of contexts and achieve substantially improved efficiency and scalability.

We have implemented CONCH on top of the SOOT framework [16] and evaluated it with 12 popular Java benchmarks and applications. Compared with kOBJ [20] and ZIPPER [22] (a representative of selective context-sensitive pointer analyses [15], [22], [24]), CONCH can speed up the two baselines together substantially (3.1x on average with a maximum of 15.9x) and analyze 7 more programs scalably, but at no loss of precision for 10 programs and only a negligible loss of precision (less than 0.1%) for the remaining two.

In summary, this paper makes the following contributions:

- We present context debloating, a new approach for accelerating all object-sensitive pointer analysis algorithms.
- We give a set of three mostly necesssary conditions for determining an object's context-dependability and propose a new lightweight IFDS-based algorithm for verifying them on the PAG representation [1] of a program.
- We have implemented CONCH in the SOOT framework and will release it soon as an open-source tool.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00018

```
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    this.g = new A(); // A
16  }
17  void foo() {
18    Object o1 = new Object(); // O1
19    A a1 = this.g;
20    a1.setF(o1);
21    Object v1 = a1.getF();
22  }
23  void bar() {
24    Object o2 = new Object(); // O2
25    A a2 = this.g;
26    a2.setF(o2);
27    Object v2 = a2.getF();
28 }}
```

Fig. 1: An example for illustrating object sensitivity.

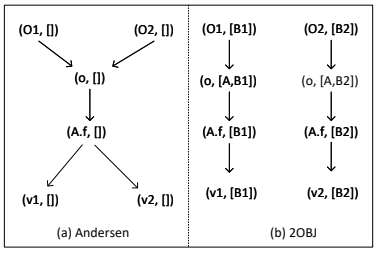


Fig. 2: Computing the points-to information for v1 and v2 in Figure 1 by applying Andersen's analysis and 2OBJ.

- We have extensively evaluated the effectiveness of CONCH (using several popular metrics) and demonstrated its practical significance for real-world programs.

The rest of this paper is organized as follows. Section II motivates our approach. Section III gives a version of kOBJ that supports context debloating. Section IV presents our CONCH approach. In Section V, we evaluate the effectiveness of CONCH in terms of context debloating. Section VI discusses the related work. Finally, Section VII concludes the paper.

II. MOTIVATION

We first review object sensitivity as a context abstraction (Section II-A). We then examine the limitations of existing object-sensitive pointer analysis algorithms (Section II-B). Finally, we motivate context debloating, by describing the basic idea behind this new approach, examining the main challenges faced in realizing it efficiently and effectively, and discussing our solution for addressing these challenges (Section II-C).

A. Object Sensitivity

We briefly review object-sensitive pointer analysis with an example given in Figure 1. In lines 7-11, we define class A, which has a field f and its corresponding setter and getter methods. In lines 12-28, we define class B, which has a field g, a constructor, and two regular methods (foo() and bar()). In foo() (bar()) of class B, an instance of java.lang.Object, O1 (O2) is created. Later, O1 (O2) is firstly stored into A.f and then loaded into v1 (v2) via the setF() and getF() methods, respectively. In main(), two instances of B, B1 and B2, are created and used as the receivers for invoking foo() and bar(), respectively.

In a context-insensitive Andersen's analysis [1], [29], every method is analyzed only once under an empty context, [ ]. Let $\overline{\mathsf{pts}}(v)$ denote the points-to set of a variable v thus computed. As illustrated in Figure 2(a), O1 and O2 are merged at o (line 9) and will later flow spuriously to v2 and v1, respectively. Hence, we have $\overline{\mathsf{pts}}(\texttt{v1}) = \overline{\mathsf{pts}}(\texttt{v2}) = \{\texttt{O1}, \texttt{O2}\}$.

In a k-object-sensitive pointer analysis (kOBJ), denoted A, the calling contexts of a method are distinguished by its receiver objects, with each being abstracted by its k-most-recent allocation sites [19], [20]. We write $\mathsf{pts}_A(v, c)$ to represent the points-to set of a variable v thus computed under a 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 20 and 26 (lines 21 and 27) under two different contexts, $[\texttt{A}, \texttt{B1}]$ and $[\texttt{A}, \texttt{B2}]$. 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 (Figure 2(b)). Hence, $\mathsf{pts}_{2\text{OBJ}}(\texttt{v1}, [\texttt{B1}]) = \{(\texttt{O1}, [\texttt{B1}])\}$ and $\mathsf{pts}_{2\text{OBJ}}(\texttt{v2}, [\texttt{B2}]) = \{(\texttt{O2}, [\texttt{B2}])\}$, without the spurious points-to information generated by Andersen's analysis.

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

B. Limitations of Existing Algorithms

We now use an example in Figure 3, which reuses class B from Figure 1, to reveal the limitations of kOBJ [19], [20] and existing approaches for selective context-sensitivity [15], [21]–[24] in analyzing real-world programs.

In lines 29-51, we define class C with a total of 2^{n+1} methods. In lines 30-38, where $0 \leqslant j < 2^{i-1}$ ($2^{i-1} \leqslant j < 2^i$), a method, $\text{foo}_{i,j}()$ ($\text{bar}_{i,j}()$), is defined, in which an object, $C_{i,j}$, is created and used as the receiver to invoke $\text{foo}_{i-1,\frac{j}{2}}()$ ($\text{bar}_{i-1,\frac{j}{2}}()$). In lines 39-51, we define $\text{foo}_{0,0}()$ ($\text{bar}_{0,0}()$), where an instance of B (defined in Figure 1), B3 (B4), is created and used to invoke foo() (bar()). In main() (lines 53-65), 2^n instances of C, denoted as $\text{C}_{n,j}$, where $0 \leqslant j < 2^n$, are created and used as the receivers to call $\text{foo}_{n-1,\frac{j}{2}}()$ when $j < 2^{n-1}$ and $\text{bar}_{n-1,\frac{j}{2}}()$ when $j \geq 2^{n-1}$.

Figure 4 depicts the OAG (Object Allocation Graph) [30], where an edge $O \rightarrow O'$ signifies that O is an allocator of O'. For kOBJ [11], [20], 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 backwards. For example, the contexts of foo() and bar() are $\{[\texttt{B3}, \text{C}_{1,\frac{j}{2^{k-2}}}, \cdots, \text{C}_{k-2,\frac{j}{2}}, \text{C}_{k-1,j}] \mid 0 \leqslant j < 2^{k-2}\}$ and $\{[\texttt{B4}, \text{C}_{1,\frac{j}{2^{k-2}}}, \cdots, \text{C}_{k-2,\frac{j}{2}}, \text{C}_{k-1,j}] \mid 2^{k-2} \leqslant j < 2^{k-1}\}$, respectively. Let $C_j(X) = [\text{A}, X, \text{C}_{1,\frac{j}{2^{k-3}}}, \cdots, \text{C}_{k-3,\frac{j}{2}}, \text{C}_{k-2,j}]$. Both setF() and getF() share the contexts in $\{C_j(\texttt{B3}) \mid 0 \leqslant j < 2^{k-3}\} \cup \{C_j(\texttt{B4}) \mid 2^{k-3} \leqslant j < 2^{k-2}\}$.

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 foo(), bar(), setF() and getF(). As k increases, such a method becomes exponentially expensive to analyze, consuming more and more memory and analysis time.

```
29 class C {
30   void foo_{i,j}() {// j < 2^{i-1}
31     C c_{i,j} = new C();// C_{i,j}
32     c_{i,j}.foo_{i-1,j/2}();
33   }
34   D bar_{i,j}(D d) {// 2^{i-1} ≤ j
35     C c_{i,j} = new C();// C_{i,j}
36     c_{i,j}.bar_{i-1,j/2}(d);
37     return d;
38   }
39   void foo_{0,0}() {
40     B b3 = new B();// B3
41     b3.foo();
42   }
43   D bar_{0,0}(D d) {
44     B b4 = new B();// B4
49     b4.bar();
50     return d;
51 }}
52 class D {}
53 void main() {
54   D d = new D;// D
55   C c = new C();// C_{n,0}
56   c.foo_{n-1,0}();
57   ...
58   C c = new C();// C_{n,2^n-1-1}
59   c.foo_{n-1,2^{n-2}-1}();
60   C c = new C();// C_{n,2^{n-1}}
61   c.bar_{n-1,2^{n-2}}(d);
62   ...
63   C c = new C();// C_{n,2^n-1}
64   c.bar_{n-1,2^{n-1}-1}(d);
65 }
```

Fig. 3: An example for motivating CONCH ($1 \leqslant i \leqslant n$ and $0 \leqslant j < 2^i$), reusing class B defined in lines 12-28 in Figure 1.

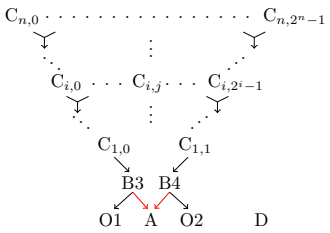


Fig. 4: The object allocation graph (OAG) for Figure 3, where only the two edges in red will remain after context debloating.

Existing approaches for selective context-sensitivity [15], [21]–[24] can improve the efficiency and scalability of kOBJ. For example, ZIPPER [22], which does not preserve the precision of kOBJ, will select `main()`, `B()`, `foo()`, `bar()`, and $\texttt{foo}_{i,j}()$ (where $j < \frac{r^i}{2}$) to be analyzed context-insensitively. However, the context explosion problem still remains for $\texttt{bar}_{i,j}()$, `setF()` and `getF()`. EAGLE [15], [25], which preserves the precision of kOBJ, is worse as it will also analyze `B()`, `foo()` and `bar()` partially context-sensitively.

C. CONCH: Our Context Debloating Approach

1) Basic Idea: We offer a new approach to mitigating the context explosion problem. Our approach, named CONCH (CONtext-dependability CHecking), aims to debloat contexts during the pointer analysis and thus complements the prior work on selective context-sensitivity. CONCH can be plugged into all object-sensitive analysis algorithms, including kOBJ and its various incarnations for supporting selective context-sensitivity [15], [21]–[24], to boost their performance significantly with negligible loss in precision. For our motivating example, only `A` is context-dependent. Handling any of the other objects context-sensitively will cost an exponential increase in analysis time without any precision benefit.

To illustrate context debloating using the OAG in Figure 4, we will remove all the allocators of a context-independent object so that the exponential growth of contexts for the object is avoided completely. Under CONCH, only the two edges in red will remain, as `A` is the only context-dependent object in the example. This implies that only `setF()` and `getF()` will be analyzed context-sensitively under `[A, B3]` and `[A, B4]`. All the other methods will be analyzed context-insensitively. For this example, debloating contexts can help kOBJ and its variants reduce their analysis times and memory consumption significantly without losing any precision.

Let A be any existing object-sensitive analysis for Java. In practice, A is usually used for analyzing a program according to client's needs under different settings (depending on, for example, how complex Java features such as exceptions, reflection, and native code are handled and whether or not certain objects are pre-configured to be context-insensitive empirically). Apparently, A will exhibit different analysis times under different settings, a problem that we do not address in this paper. However, for a fixed setting given, CONCH can accelerate A at no or little loss of precision by debloating its contexts. Let A_1 and A_2 be two different object-sensitive analyses used for analyzing a program under two different settings S_1 and S_2. It is possible that A_1 is faster than A_2 under S_1 but the opposite is true under S_2, which is again a problem that we do not investigate here. However, for a fixed setting given, CONCH can accelerate both analyses at no or little loss of precision by debloating their contexts.

2) Challenges: To debloat contexts, we must find context-dependent objects. Recently, the following two necessary conditions are given for determining the context-dependability of an object O allocated in a method m based on CFL reachability, requiring us to check the existence of a write into and a read from an access path $O.f_1.\cdots.f_n$ context-sensitively (where the two accesses often happen outside m) [15], [25]:

- $A \xrightarrow{(_c} O.f_1.\cdots.f_n$: 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
- $O.f_1.\cdots.f_n \xrightarrow{)_c} v$: there exists a load of $O.f_1.\cdots.f_n$ flowing into a variable v outside m under also c.

where context matching is formulated by solving the standard balanced parentheses problem [28]. If these two conditions hold, O must be context-dependent. Otherwise, different objects A flowing into $O.f_1.\cdots.f_n$ under different calling contexts of m will be conflated, causing them to flow into different variables v spuriously. In object-sensitive pointer analysis, the parameters and return variable of a method are also conceptually regarded as special fields of its receiver objects [15], [25]. Thus, in the access path above, a field f_i can be either a real Java field or one of such special fields.

Unfortunately, verifying these two conditions precisely is undecidable [31], as it requires us to solve kOBJ fully context-sensitively (with $k = \infty$). In addition, weakening these two conditions [15], [25] will over-approximate unduly the number of context-dependent objects found but approximating them heuristically [21]–[24] may cut it down significantly but at the expense of some significant precision loss.

3) Our Solution: To identify context-dependent objects efficiently and effectively, our key insight is to approximate the two aforementioned necessary conditions with the three conditions that are linearly verifiable (in terms of the number

of statements) and mostly necessary for real code, based on three key observations governing how objects are used.

Like the prior work on selective context-sensitivity [21]–[24], CONCH also relies on the points-to information, $\overline{\mathsf{pts}}$, pre-computed by Andersen's analysis.

Observation 1. *A context-dependent object O often has at least one instance field $O.f$ that is both written into ($x.f = \cdots$) and read from ($\cdots = x.f$), where $O \in \overline{\mathsf{pts}}(x)$.*

```
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. 5: A context-dependent object `B` violating Obs 1.

There can be rare cases, as illustrated in Figure 5, where Obs 1 may not be valid for some context-dependent objects, such as `B`. Under object-sensitivity [15], [25], `O` pointed to by `p` is first written into `B.q` and then returned and stored into `v`. As discussed in Section II-C2, `q` is considered as a special field of `B`. Such cases are rare in real-world object-oriented programs, as CONCH loses little precision (Section V).

Observation 2. *A context-dependent object O, pointed to by a variable or a field of some object according to $\overline{\mathsf{pts}}$, usually flows out of its containing method (for allocating O).*

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

(a) Case 1 from `Vector`

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

(b) Case 2 from `HashMap`

```
1 void SunJCE_e_a(...) {
2   BufferedReader br = new BufferedReader();
3   this.f = new StreamTokenizer(br);
4 }
```

(c) Case 3 from `SunJCE_e`

Fig. 6: Three common cases abstracted from JDK for Obs 2.

Figure 6 gives three representative cases abstracted from the JDK where Obs 2 holds. In Figure 6(a), the array object created flows out of the constructor via a store. In Figure 6(b), the `KeyIterator` object created flows out of `iterator()` directly via a return. In Figure 6(c), we have a slightly more complicated case. The `BufferedReader` object created flows out of its containing method as it is stored into the `input` field of the `StreamTokenizer` object, which flows out of the containing method via a store. The objects that cannot flow out of their containing methods are usually context-independent as they are often created and used locally.

Observation 3. *A context-dependent object O tends to have a store statement $x.f = y$ in a method m', where $O \in \overline{\mathsf{pts}}(x)$. Let m be the method where O is allocated if m' is a constructor (i.e., the constructor for creating O) and m' otherwise. Then y **(a)** is data-dependent on a parameter of m or **(b)** points to a context-dependent object.*

```
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; this.value = v;
6 }
```

(b) Case 2 from `HashMap`.

```
1 HashSet() {
2   this.map = new HashMap();
3 }
```

```
4 HashMap(...) {
5   this.table = new Entry[10];
6 }
```

(c) Case 3 from `HashSet` and `HashMap`.

Fig. 7: Three common cases abstracted from JDK for Obs 3.

Figure 7 gives three representative cases abstracted from the JDK where Obs 3 holds. In Figure 7(a), O is the `Object[]` object allocated in line 2 and $x.f = y$ is `this.elems[idx] = e`, which is modeled as `this.elems.arr = e`, where arr is a special field introduced to represent all the elements of an array (Section III). In this case, $m = m' =$ `set()`. Here, `e` satisfies Obs 3(a) trivially. In Figure 7(b), O is the `Entry` object allocated in line 2, $x.f = y$ is `this.key = k`/`this.value = v`, $m' =$ `Entry()`, and $m =$ `addEntry()`. Here, `k`/`v` (in line 5) also satisfies Obs 3(a) trivially. In Figure 7(c), O is the `HashMap` object allocated in line 2, $x.f = y$ is `this.table = new Entry[10]`, $m' =$ `HashMap()`, and $m =$ `HashSet()`. As `new Entry[10]` is context-dependent by Obs 2 (as well as Obs 1 and Obs 3 if the entire code is considered), the `HashMap` object in line 2 is also context-dependent by Obs 3(b). In Obs 3(b), the circular dependences on context-dependability are solved optimistically in Algorithm 1.

4) Motivating Example: For this example given in Figure 3 (with class `B` from Figure 1), CONCH will identify `A` as the only context-dependent object. Let us examine Figure 1, where `A` is created in line 15. `A` is context-dependent as it satisfies all the three observations: (1) `A` has an instance field `f`, which has a write and a read in lines 9 and 10, respectively (Obs 1), (2) `A` can flow out of `B()` via the store statement in line 15 (Obs 2), and (3) `o` is stored into `A.f` in line 9, where `o` happens to be a parameter of `setF()` (Obs 3). Let us now consider `B3` and `B4` created in Figure 3. Both are context-independent as both satisfy Obs 1 (with an instance field `g` of `B3`/`B4` stored in `B()` and loaded in `foo()`/`bar()` in Figure 1) and Obs 3 (due to the existence of `this.g = new A()// A` in line 15, where `A` is context-dependent) but not Obs 2 (as `B3`/`B4` does not flow out of its containing method $\texttt{foo}_{0,0}\texttt{()}/\texttt{bar}_{0,0}\texttt{()}$). Finally, all the other objects are context-independent as they do not contain instance fields and are used only locally, failing to satisfy any of the three observations stated.

5) Discussion: CONCH relies on Obs 1– Obs 3 to generate three corresponding linearly verifiable conditions for determining the context-dependability of an object. In Section IV, we introduce a lightweight IFDS-based algorithm for verifying these conditions efficiently. In Section V, we demonstrate CONCH is highly effective for real-world programs.

III. kOBJ WITH CONTEXT DEBLOATING

We formalize context debloating here. We first review the classic algorithm for kOBJ (Section III-A) and then adapt it

to support context debloating (Section III-B). CONCH can be used similarly for debloating contexts for any variant of kOBJ.

A. kOBJ

We describe kOBJ [11], [19], [20], [32] by considering a simplified subset of Java, with five types of labeled statements in Table I. Note that "x = **new** $T(...)$" is modeled as "x = **new** T; x.$\langle$init$\rangle$(...)", where $\langle$init$\rangle$(...) is the corresponding constructor invoked. The control flow statements are irrelevant since kOBJ is context-sensitive but flow-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 V discusses how to handle static method calls and other complex language features such as exceptions, reflection, and native code.

Kind	Statement	Kind	Statement
NEW	$l : x =$ **new** T	ASSIGN	$l : x = y$
STORE	$l : x.f = y$	LOAD	$l : x = y.f$
CALL	$l : x = a_0.f(a_1, ..., a_r)$		

TABLE I: Five types of statements analyzed by kOBJ.

kOBJ makes use of the following domains: $\mathbb{V}$, $\mathbb{H}$, $\mathbb{M}$, $\mathbb{F}$, and $\mathbb{L}$, which represent 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 $ctx = [e_1, \cdots, e_n] \in \mathbb{C}$ and a context element $e \in \mathbb{H}$, we write $e +\!\!+ ctx$ for $[e, e_1, \cdots, e_n]$ and $\lceil ctx \rceil_k$ for $[e_1, \cdots, e_k]$.

The following auxiliary functions are also used:

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

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 8 gives the five rules used by kOBJ for analyzing the five kinds of statements in table I. 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 handled in the standard manner. In [CALL], a call to an instance method $x = a_0.f(a_1, ..., a_r)$ is analyzed. In this paper, we write $this^{m'}$, $p_i^{m'}$ and $ret^{m'}$ for the "this" variable, i-th parameter and return variable of m', respectively, where m' is a target method resolved. 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") $= \{[]\}$.

B. Context Debloating

To debloat contexts, we assume that $\mathcal{D}$ represents the set of context-independent objects found by CONCH. Thus, the

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

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

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

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

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

Fig. 8: Rules for kOBJ.

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

Fig. 9: Adapting [NEW] to support context debloating.

objects in $\mathbb{H} \setminus \mathcal{D}$ are context-dependent. To modify kOBJ to support context debloating, we simply replace [NEW] by [NEW+D] given in Figure 9. For a context-dependent object, we proceed identically as before. For a context-independent object, we no longer distinguish it under its different allocators, by setting its heap context as $hctx = []$, eliminating the context explosion problem that would otherwise have occurred when it is used as a receiver object of an invoked method.

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

IV. CONCH: OUR CONTEXT DEBLOATING APPROACH

We introduce an IFDS-based algorithm [28] for verifying efficiently the three mostly necessary conditions stated in Obs 1 – Obs 3 to find the context-dependent objects in a program. As these conditions are not sufficient, we may mis-classify context-independent objects as being context-dependent (but err on the side of preserving precision). As these conditions are mostly but not strictly necessary (Figure 5), we may occasionally mis-classify context-dependent objects as being context-independent (at a small loss of precision). We use the points-to information $\overline{\text{pts}}$ pre-computed by Andersen's analysis [29] (which is the context-insensitive version of Figure 8). We first give a high-level overview of Algorithm 1 and then discuss how to verify these conditions.

CONCH takes a program P as input and returns $\mathcal{D}$ as the set of context-independent objects in P for context-debloating. Some additional notations 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 $O \in \overline{\text{pts}}(x)$. CI and CD, which are initialized to be $\emptyset$ (line 1), represent the sets

Algorithm 1: CONCH: context debloating.

```
   Input: P                                   // Input program
   Output: D.                  // Set of Context-Indep Objects
1  CI ← CD ← ∅
2  for O_l ∈ ℍ do
3      if ∄ f ∈ fieldsOf(O_l) s.t hasLoad(O_l, f)∧hasStore(O_l, f) then
4          CI = CI ∪ {O_l}                                  // Obs 1
5      else if O_l ∉ leakObjects then
6          CI = CI ∪ {O_l}                                  // Obs 2
7      else
8          R(O_l) = {l' : x.f = y in P | O_l ∈ pts(x)}
9          for l' : x.f = y ∈ R(O_l) do
10             if methodOf(l') is a constructor of O_l then
11                 m = methodOf(l)
12             else
13                 m = methodOf(l')
14             if depOnParam(y, m) then
15                 CD = CD ∪ {O_l}                          // Obs 3(a)
16                 break
17 UK ← ℍ \ (CI ∪ CD), changed ← true
18 while changed do
19     changed ← false
20     for O_l ∈ UK do
21         if ∃ l' : x.f = y ∈ R(O_l) s.t. pts(O_l.f) ∩ CD ≠ ∅ then
22             CD = CD ∪ {O_l}                              // Obs 3(b)
23             changed ← true
24 D = CI ∪ (UK \ CD);
25 return D
```

of context-independent and context-dependent objects found so far, respectively. There are two stages, with the first stage (lines 2-16) for verifying Obs 1, Obs 2 and Obs 3(a) and the second stage (lines 17-23) for verifying Obs 3(b).

A. Verifying Observation 1

In lines 3-4, an object O_l is classified as being context-independent (and inserted into CI) if it does not satisfy Obs 1. Otherwise, we will proceed to verify Obs 2 and Obs 3.

B. Verifying Observation 2

In lines 5-6, an object O_l is classified as being context-independent (and inserted into CI) if it does not satisfy Obs 2, i.e., $O_l \notin$ leakObjects, where leakObjects contains the set of objects that can flow out of their containing methods by Obs 2. Otherwise, we will proceed to verify Obs 3.

We introduce an IFDS-based algorithm given in Figure 14 for computing leakObjects in P context-sensitively, based on the DFA (Deterministic Finite Automaton) given in Figure 13. Computing leakObjects entails reasoning about object reachability in P. Let us describe it incrementally.

Initially, we start with a parameterless method containing no calls. Its PAG (Pointer Assignment Graph) [1] can be built by the rules in Figure 10. Our analysis is field-insensitive, as reflected by [P-LOAD] and [P-STORE]. Figure 12(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 forwards in the PAG), B (moving backwards in the PAG), and E (exiting from the allocating method). Due to the absence of parameters and returns, no

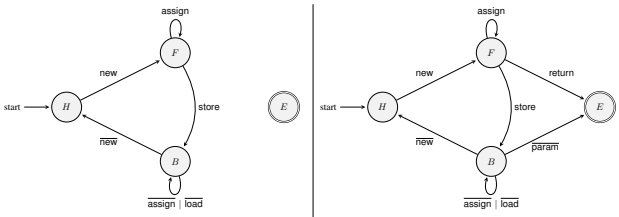
$$\frac{l : x = \textbf{new}\ T}{O_l \xrightarrow{\text{new}} x \quad x \xrightarrow{\overline{\text{new}}} O_l}\ [\text{P-NEW}] \qquad \frac{x = y}{y \xrightarrow{\text{assign}} x \quad x \xrightarrow{\overline{\text{assign}}} y}\ [\text{P-ASSIGN}]$$

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

Fig. 10: PAG edges for a parameterless method with no calls.

$$\frac{p \text{ is a parameter}}{p \xrightarrow{\overline{\text{param}}} p}\ [\text{P-PARAM}] \qquad \frac{ret \text{ is a return variable}}{ret \xrightarrow{\text{return}} ret}\ [\text{P-RETURN}]$$

Fig. 11: PAG edges for parameters and return variables.



(a) No calls/parameters/returns (b) No calls

Fig. 12: Two intermediate DFAs for the DFA in Figure 13.

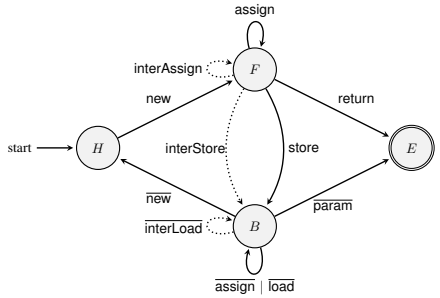


Fig. 13: The DFA for verifying Obs 2.

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 13(a)). 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 firstly stored into an access path of some locally allocated object O' and then 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 (but are used by the DFA in Figure 15 for computing depOnParam). In addition, the DFA also ignores the 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 fails to hold, CONCH may classify a context-dependent object as being context-independent, causing the underlying pointer analysis to lose some precision.

To support parameters and return variables, we add their self-loop edges using the rules in Figure 11 and transform the DFA in Figure 12(a) into the one in Figure 12(b). Once an object allocated in a method flows to a parameter (suggested by $\overline{\mathsf{param}}$) or the return variable (suggested by return) under state E, it has leaked.

The final DFA is presented in Figure 13, where the three dotted transitions are added for handling call statements. While each method has its own PAG, some summary edges are added to its PAG for its call sites to capture the inter-procedural value-flows across these call sites context-sensitively, along the three dotted transitions. The call graph is built using $\overline{\mathsf{pts}}$.

Given a call statement $l : x = a_0.f(a_1, ..., a_r)$ contained in method m, let m' be a resolved target method (with $p_i^{m'}$ being its i-th parameter and $ret^{m'}$ being its return variable). Let n_1 and n_2 be two PAG nodes. We write $\langle n_1, S_1\rangle \rightarrow \langle n_2, S_2\rangle$ (known as a path edge in [28]) to indicate that node n_1 at state S_1 can reach node n_2 at state S_2. Let us write G_m as the PAG of m. There are four cases considered when 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., $p_j^{m'}.f_1.\cdots.f_n = p_i^{m'}$. Thus, we add a summary edge, $a_i \xrightarrow{\mathsf{interStore}} a_j$ (i.e., $a_j.f = a_i$), 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. Thus, we add a summary edge, $a_i \xrightarrow{\mathsf{interAssign}} x$, 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., $ret^{m'} = p_i^{m'}.f_1.\cdots.f_n$. Thus, we add a summary edge, $x \xrightarrow{\mathsf{interLoad}} a_i$ (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. We introduce a symbolic object Sym_l to abstract all the possible objects returned from the call site l and continue our analysis in m.

Figure 14 gives our IFDS-based algorithm [28] for computing $\mathsf{leakObjects}$, operating on a PAG instead of a CFG representation of a program. The rules in [SEEDS] inject three kinds of path edges, where the first one is for tracing leak objects while the other two are for finding summary edges (which are not injected on-demand in order to improve parallelism in a parallel implementation of our algorithm). The rules in [PROPAGATE] perform the reachability analysis according to the DFA in Figure 13. Note that the three dotted transitions in the DFA are implicitly handled by the summary edges generated in [SUMMARY]. Finally, we collect the objects that can reach the final state, E, by using [COLLECT].

C. Verifying Observation 3

In lines 8-16, we verify if an object O_l satisfies Obs 3(a). In the case of a positive answer, O_l is considered immediately as being context-dependent (and thus inserted into CD), since O_l has already satisfied both Obs 1 and Obs 2 at this point. Otherwise, we proceed to verify Obs 3(b) in lines 17-23.

The key to verifying Obs 3(a) lies in $\mathsf{depOnParam}(y, m)$, which returns true if y is data-dependent on any parameter of method m. We have also designed and implemented an IFDS-based algorithm for computing $\mathsf{depOnParam}$, in a similar manner as how we have computed $\mathsf{leakObjects}$ in Figure 14, by making use of a simpler DFA given in Figure 15.

This DFA has only two states, F and E, recognizing only four types of PAG edges, where $\mathsf{interAssign}$ is a summary edge introduced for supporting call statements. Given a call statement $l : x = a_0.f(a_1, \cdots, a_r)$ 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, $a_i \xrightarrow{\mathsf{interAssign}} x$, to the PAG of m to propagate this reachability fact inter-procedurally from the callee m' to the caller m.

Our algorithm for computing $\mathsf{depOnParam}$, which proceeds forwards from method parameters, is a simplified version of the one in Figure 14. For [SEEDS], only the parameters need to be injected. The rules for [PROPAGATE], are similar. For [SUMMARY], we use the summary edges added as discussed above. Finally, let $\mathsf{dps}(v, m_v) = \{p_i^{m_v} \mid \langle p_i^{m_v}, F\rangle \rightarrow \langle y, F\rangle\}$, where v is a variable defined in its containing method m_v and $p_i^{m_v}$ is some (i-th) parameter of m_v. Then $\mathsf{depOnParam}(y, m)$ can be defined recursively as (by taking care of chained constructors, in practice):

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

where a_i is the corresponding argument of $p_i^{m_y}$.

Finally, Obs 3(b) can be verified straightforwardly. At this point, CI and CD contain the sets of context-independent and context-dependent objects found so far. Let O be an object in $\mathbb{H} \setminus (\mathrm{CI} \cup \mathrm{CD})$. O is regarded as being context-dependent if it can point to any context-dependent object (found so far) transitively and context-independent otherwise.

D. Soundness and Time Complexity

CONCH is sound as it may mis-classify some context-dependent objects as being context-independent and thus cause the underlying pointer analysis to produce over-approximated points-to information, resulting in some loss of precision.

The worst-case time complexity of CONCH in analyzing a program P is linear to the number of its statements, for three reasons. First, $\mathsf{leakObjects}$ can be computed according to Figure 14 in $O(ED^3)$ [28], where E is the number of PAG edges in P, which are constructed linearly to the number of statements in P according to Figures 10 and 11, and $D = 4$ is the number of states of the DFA in Figure 13. Second, the first stage of Algorithm 1 (lines 2-16) runs in $O(|\mathbb{L}|)$,

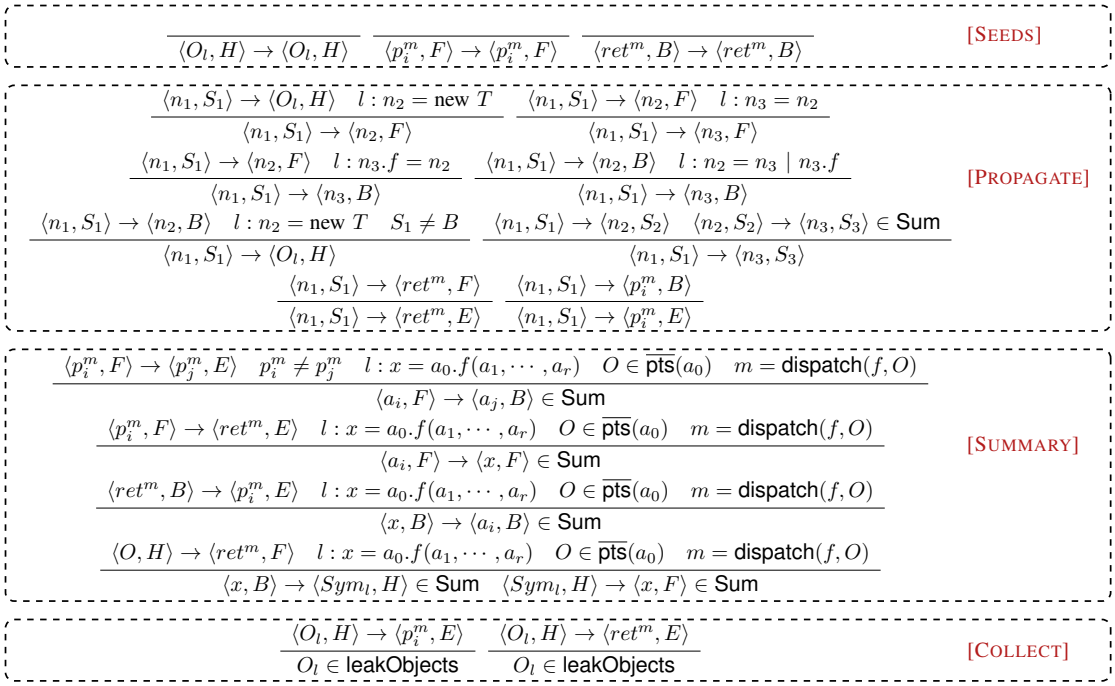


Fig. 14: Rules for computing leakObjects, i.e., the set of objects that can flow out of their containing methods for verifying Obs 2. $S_i \in \{H, F, B\}$, where $i \in \{1, 2, 3\}$ and Sym_l is a symbolic object abstracting all objects returned from call site l.

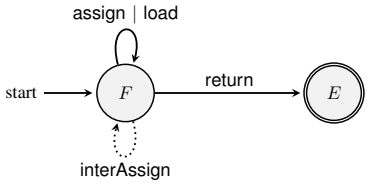


Fig. 15: The DFA used for computing depOnParam.

where $\mathbb{L}$ is the set of statements in P. Finally, the second stage of Algorithm 1 (lines 17-23) can be efficiently performed in $O(|\mathbb{H}|)$, where $\mathbb{H}$ is the set of heap objects in P.

V. Evaluation

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

- **RQ1.** Is Conch precise and efficient?
- **RQ2.** Can Conch speed up existing object-sensitive analysis algorithms significantly?

Implementation. We have implemented Conch in Soot [16], a program analysis and optimization framework for Java, on top of its context-insensitive Andersen's pointer analysis, Spark [1] (for computing $\overline{\text{pts}}$). Conch is implemented in about 1500 lines of Java code, which will soon be released as an open-source tool at http://www.cse.unsw.edu.au/~corg/conch along with a reproducible artifact in the form of a Docker image. As described in Section II, Conch aims to boost the performance of all object-sensitive pointer analysis algorithms. We report and analyze our results by applying Conch to debloat two representative baselines, $k$obj (an object-sensitive version of Spark) and Zipper [22] (the latest version b83b038, which can deliver the arguably best speedups for $k$obj among the recent algorithms for supporting selective context-sensitivity [15], [21]–[24] in our experimental setting). Due to space limitations, we will only summarize the significant performance benefits also achieved by Conch in debloating contexts for Eagle [15].

Experimental Setting. $k$obj is a standard in-house implementation of Spark in Soot [33]. As for Zipper (originally released in Doop [17] but used here to accelerate $k$obj in Soot), we have used an analysis setting that is as close as possible to the one used by Zipper 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 [34]. 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 [35] as is often done in the pointer analysis literature [11], [12], [22], [24]. 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 [36] and WALA [18].

We have conducted our experiments on an Intel(R) Xeon(R) CPU E5-1660 3.2GHz machine with 256GB of RAM. We

have selected a set of 12 popular Java programs, including 9 benchmarks from DaCapo [37], and 3 Java applications (`checkstyle`, `JPC` and `findbugs`). The Java library used is `jre1.6.0_45`. These are the standard Java programs that are frequently used for evaluating pointer analysis algorithms [11], [12], [22], [24]. The time budget used for running each pointer analysis on a program is set as 12 hours. The analysis time of a program is an average of three runs.

A. RQ1: Is CONCH Precise and Efficient?

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:

$$\Delta = \frac{\text{Metric(Base+D)} - \text{Metric(Base)}}{\text{Metric(Base)}} \quad (2)$$

where Metric(Base) and Metric(Base+D) are the metric numbers obtained by Base and Base+D, respectively. We use four common metrics for measuring the precision of a context-sensitive pointer analysis [11], [13], [15], [22]: (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 II gives our main results. For kOBJ, ZkOBJ denotes the version of kOBJ with selective context-sensitivity provided by ZIPPER. All the baselines (where $k \in \{2, 3\}$) and their debloated versions are compared over the 12 Java programs.

CONCH is very precise in terms of supporting context debloating while losing negligible precision. Our approach preserves the precision of all the baselines for 10 programs consisting of the 9 DaCapo benchmarks and `findbugs`. For `checkstyle` and `JPC`, our approach suffers from an average precision loss of only less than 0.1% (across the four metrics). This happens since a `PropertyChangeEvent` object created in method `firePropertyChange(...)` of class `java.beans.PropertyChangeSupport` and a `LineReader` object created in method `load(InputStream)` of `java.util.Properties` have been misclassified as being context-independent by CONCH as they do not satisfy Obs 2.

CONCH is also highly efficient (as a pre-analysis). Table III gives the times spent by SPARK [1], ZIPPER [22] and CONCH. 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 and SPARK across all the 12 programs. On average, we have 2.6 seconds (CONCH), 10.4 seconds (ZIPPER) and 12.5 seconds (SPARK). Thus, CONCH is efficient enough for supporting context debloating.

B. RQ2: Can CONCH Speed Up Baseline Analyses?

Table II also gives the analysis times of all the analyses. CONCH deliver significant speedups (geometric means) over all the baselines. For kOBJ, the speedups of 2OBJ+D over 2OBJ range from 1.7x (for `pmd`) to 9.0x (for `findbugs`) with an average of 2.6x. When $k = 3$, the speedups of 3OBJ+D over 3OBJ are more impressive, ranging from 2.9x (for `luindex`) to 9.0x (for `JPC`) with an average of 5.2x. For ZIPPER, the speedups of Z2OBJ+D over Z2OBJ range from 1.0x (for `xalan`) to 2.7x (for `antlr`) with an average of 1.8x. When $k = 3$, the speedups of Z3OBJ+D over Z3OBJ are also more impressive, ranging from 2.2x (for `xalan`) to 15.9x (for `lusearch`) with an average of 5.6x.

TABLE II: Main results. In all metrics (except for speedups), smaller is better. Given an analysis Base, Base+D is its debloated version by CONCH. OoM stands for "Out of Memory".

		Classic kOBJ				Selective kOBJ			
Prog	**Metrics**	**2OBJ**	**2OBJ+D**	**3OBJ**	**3OBJ+D**	**Z2OBJ**	**Z2OBJ+D**	**Z3OBJ**	**Z3OBJ+D**
antlr	Time (s)	45.4	13.9 (3.3x)	1049.3	185.4 (5.7x)	20.8	7.8 (2.7x)	337.9	32.1 (10.5x)
	#fail-cast	509	509	449	449	559	559	507	507
	#call-edges	51176	51176	51149	51149	51394	51394	51367	51367
	#poly-calls	1622	1622	1615	1615	1643	1643	1636	1636
	#reach-mtds	7804	7804	7803	7803	7842	7842	7841	7841
bloat	Time (s)	743.8	359.5 (2.1x)	> **12h**	4093.7	519.9	279.7 (1.9x)	OoM	2771.2
	#fail-cast	1314	1314	-	1221	1368	1368	-	1279
	#call-edges	56699	56699	-	56464	57192	57192	-	57036
	#poly-calls	1695	1695	-	1675	1732	1732	-	1716
	#reach-mtds	9021	9021	-	9005	9093	9093	-	9085
chart	Time (s)	253.0	85.5 (3.0x)	OoM	4215.9	34.6	20.3 (1.7x)	573.6	178.3 (3.2x)
	#fail-cast	1348	1348	-	1241	1418	1418	1323	1323
	#call-edges	72457	72457	-	72023	73123	73123	72738	72738
	#poly-calls	2032	2032	-	2008	2060	2060	2040	2040
	#reach-mtds	15143	15143	-	15113	15269	15269	15247	15247
eclipse	Time (s)	> **12h**	4113.5	OoM	OoM	2956.3	2487.7 (1.2x)	OoM	OoM
	#fail-cast	-	3215	-	-	3357	3357	-	-
	#call-edges	-	145763	-	-	146492	146492	-	-
	#poly-calls	-	8720	-	-	8737	8737	-	-
	#reach-mtds	-	19916	-	-	19985	19985	-	-
fop	Time (s)	18.6	10.5 (1.8x)	572.3	177.8 (3.2x)	9.2	5.1 (1.8x)	113.3	28.1 (4.0x)
	#fail-cast	395	395	336	336	444	444	400	400
	#call-edges	34120	34120	34100	34100	34343	34343	34323	34323
	#poly-calls	808	808	802	802	832	832	826	826
	#reach-mtds	7582	7582	7582	7582	7620	7620	7620	7620
luindex	Time (s)	19.4	8.7 (2.2x)	555.3	192.6 (2.9x)	9.4	4.9 (1.9x)	129.5	31.0 (4.2x)
	#fail-cast	394	394	340	340	448	448	398	398
	#call-edges	33495	33495	33468	33468	33728	33728	33701	33701
	#poly-calls	918	918	911	911	944	944	937	937
	#reach-mtds	7017	7017	7016	7016	7057	7057	7056	7056
lusearch	Time (s)	30.4	11.8 (2.6x)	2225.7	252.1 (8.8x)	13.2	5.2 (2.5x)	622.7	39.2 (15.9x)
	#fail-cast	409	409	357	357	466	466	418	418
	#call-edges	36377	36377	36350	36350	36605	36605	36578	36578
	#poly-calls	1116	1116	1109	1109	1143	1143	1136	1136
	#reach-mtds	7669	7669	7668	7668	7707	7707	7706	7706
pmd	Time (s)	41.6	24.2 (1.7x)	1236.1	257.0 (4.8x)	23.9	14.9 (1.6x)	344.7	52.5 (6.6x)
	#fail-cast	1432	1432	1367	1367	1514	1514	1461	1461
	#call-edges	59864	59864	59805	59805	60029	60029	59970	59970
	#poly-calls	2357	2357	2351	2351	2382	2382	2376	2376
	#reach-mtds	11841	11841	11841	11841	11880	11880	11880	11880
xalan	Time (s)	565.3	298.2 (1.9x)	OoM	1632.1	230.7	227.5 (1.0x)	2487.7	1125.6 (2.2x)
	#fail-cast	600	600	-	546	657	657	609	609
	#call-edges	46653	46653	-	46621	46842	46842	46815	46815
	#poly-calls	1613	1613	-	1606	1636	1636	1629	1629
	#reach-mtds	9659	9659	-	9657	9701	9701	9700	9700
checkstyle	Time (s)	1014.6	349.1 (2.9x)	> **12h**	OoM	404.4	236.1 (1.7x)	OoM	4887.4
	#fail-cast	1130	1130	-	-	1206	1206	-	1117
	#call-edges	67039	67041	-	-	67854	67854	-	66892
	#poly-calls	2210	2210	-	-	2268	2268	-	2211
	#reach-mtds	12314	12314	-	-	12383	12383	-	12342
JPC	Time (s)	106.1	54.6 (1.9x)	2163.3	240.6 (9.0x)	34.4	26.3 (1.3x)	181.0	44.8 (4.0x)
	#fail-cast	1356	1356	1206	1206	1431	1431	1278	1278
	#call-edges	80965	80978	79297	79310	81616	81629	79932	79945
	#poly-calls	4263	4264	4127	4128	4324	4325	4187	4188
	#reach-mtds	15508	15508	15161	15161	15582	15582	15232	15232
findbugs	Time (s)	1629.6	180.1 (9.0x)	OoM	936.3	131.3	50.3 (2.6x)	1890.0	186.8 (10.1x)
	#fail-cast	2072	2072	-	1696	2144	2144	1956	1956
	#call-edges	87915	87915	-	86993	88567	88567	87741	87741
	#poly-calls	3655	3655	-	3621	3670	3670	3643	3643
	#reach-mtds	16266	16266	-	16219	16315	16315	16287	16287

TABLE III: Times spent by pre-analyses in seconds.

	antlr	bloat	chart	eclipse	fop	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs
SPARK	8.6	8.9	15.6	25.9	7.6	6.9	7.8	11.7	8.7	14.0	17.4	16.7
ZIPPER	4.6	6.4	16.4	25.5	4.0	3.7	4.3	9.5	10.2	14.5	9.8	16.2
CONCH	1.7	2.0	3.2	6.6	1.5	1.4	1.4	2.2	2.5	2.9	2.6	3.1

These results suggest that the speedups delivered by CONCH increase as k increases, implying that CONCH can help all the baselines improve their scalability. In particular, 2OBJ+D scales one more benchmark, i.e., `eclipse` than 2OBJ, 3OBJ+D can scale 4 more benchmarks (`bloat`, `chart`, `xalan`, and `findbugs`) than 3OBJ, and Z3OBJ+D can scale 2 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**").

Due to space limitations, let us summarize briefly the significant performance benefits achieved by CONCH in debloating contexts for EAGLE [15], [25]. Unlike ZIPPER (which makes kOBJ run faster while losing precision), EAGLE aims to accelerate kOBJ while preserving its precision. For kOBJ, let EkOBJ be the version of kOBJ with selective context-sensitivity enabled by EAGLE. For the four precision metrics considered in Table II, *#fail-cast*, *#call-edges*, *#poly-calls*, and *#reach-mtds*, EkOBJ yields the same results as kOBJ and EkOBJ+D yields the same results as kOBJ+D in theory. As for the performance speedups achieved, CONCH is nearly as effective for EAGLE as for ZIPPER. The speedups of E2OBJ+D over E2OBJ range from 1.3x (for `eclipse`) to 5.7x (for `findbugs`) with an average of 2.1x, and the speedups of E3OBJ+D over E3OBJ range from 2.8x (for `luindex`) to 9.2x (for `lusearch`) with an average of 4.8x. In addition, CONCH scale 4 more benchmarks, `bloat`, `chart`, `xalan`, and `findbugs`, under E3OBJ+D than under E3OBJ.

Therefore, CONCH can accelerate existing object-sensitive pointer analyses significantly with negligible loss in precision. These include not only kOBJ (the standard algorithm) but also its variants enabled by, e.g., ZIPPER [22] and EAGLE [15], [25] (the two recent attempts on applying selective context-sensitivity to improve the performance of kOBJ).

Below we analyze in detail why context debloating can enable baseline analyses, kOBJ and ZkOBJ, to improve their efficiency and scalability (as reported in Table II).

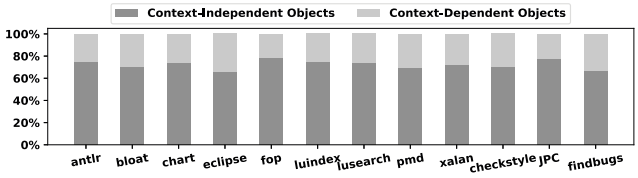


Fig. 16: Percentage distribution of the two types of objects.

Figure 16 depicts the percentage distribution of context-dependent objects 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 72.4%. Thus, a large amount of precision-irrelevant contexts has been eliminated via context debloating.

Table IV compares the baseline analyses (i.e., kOBJ and ZkOBJ) and their debloated counterparts (i.e., kOBJ+D and ZkOBJ+D) in terms of the average number of contexts analyzed for a method, where $k \in \{2, 3\}$. The debloated analyses

TABLE IV: Average number of contexts analyzed for a method by kOBJ, kOBJ+D, ZkOBJ and ZkOBJ+D, where $k \in \{2, 3\}$.

	antlr	bloat	chart	eclipse	fop	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs
2OBJ	27.1	30.3	36.5	-	15.6	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	9.9	10.3	10.3	31.6	41.4	16.0	18.8
Z2OBJ	8.1	14.3	6.9	14.8	4.9	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	4.0	4.0	4.4	11.8	14.4	6.1	7.3
3OBJ	99.8	-	-	-	53.0	58.1	91.5	53.5	-	-	87.2	-
3OBJ+D	24.6	39.0	78.9	-	19.1	21.3	22.2	18.7	61.6	-	26.1	29.5
Z3OBJ	26.5	-	21.7	-	14.3	16.7	23.5	17.6	60.7	-	14.5	25.7
Z3OBJ+D	8.2	17.3	12.5	-	6.4	7.1	7.3	7.1	22.6	57.5	7.7	10.8

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 a substantial reduction it has achieved in the number of context-sensitive facts inferred. In Table V, #cs-gpts, #cs-pts and #cs-fpts represent the numbers of context-sensitive objects pointed by global variables (i.e., static fields), local variables and instance fields, respectively, and #cs-calls represents 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 is significantly less than 2OBJ. Similarly, Z3OBJ+D is also much faster than Z3OBJ for `lusearch`. However, the analysis time of a pointer analysis is not linearly proportional to the number of context-sensitive facts computed [13]. 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.

VI. 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 object-sensitive pointer analysis (i.e., kOBJ) [15], [22], [24]. EAGLE [15] improves the efficiency of kOBJ while preserving its precision by conservatively reasoning about value flows via CFL reachability. ZIPPER [22], as a representative of non-precision-preserving approaches [21]–[24], [38], trades precision for efficiency by exploiting several value flow patterns. These techniques mitigate the context explosion problem of kOBJ 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 a only negligible loss of precision.

When applying CONCH to debloat contexts for ZkOBJ [22] and EkOBJ [15], we have observed that their relative performance advantages depend on the experimental settings used (for good reasons). In [15], EkOBJ outperforms ZkOBJ when both are evaluated in SOOT (by using the exception analysis provided by SPARK [1] and disallowing manual

TABLE V: Context-sensitive facts.

Prog	Metrics	Classic kOBJ				Selective kOBJ			
		2OBJ	2OBJ+D	3OBJ	3OBJ+D	Z2OBJ	Z2OBJ+D	Z3OBJ	Z3OBJ+D
antlr	#cs-gpts	4.9K	2.1K	12.1K	2.5K	5.7K	2.3K	17.6K	2.7K
	#cs-pts	19.8M	3.6M	228.8M	32.1M	18.6M	3.3M	205.1M	10.4M
	#cs-fpts	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	1.9M	0.5M	22.7M	1.1M
	Total	25.8M	5.1M	329.8M	61.1M	21.1M	3.9M	241.6M	17.8M
bloat	#cs-gpts	3.1K	1.9K	-	2.3K	3.9K	2.0K	-	2.4K
	#cs-pts	159.8M	68.0M	-	325.0M	140.9M	53.6M	-	235.0M
	#cs-fpts	5.7M	4.6M	-	28.8M	6.9M	4.6M	-	28.0M
	#cs-calls	47.1M	20.9M	-	112.0M	38.2M	16.4M	-	74.0M
	Total	212.7M	93.5M	-	465.8M	186.0M	74.5M	-	336.9M
chart	#cs-gpts	12.5K	6.9K	-	11.3K	10.1K	5.5K	24.6K	6.9K
	#cs-pts	56.9M	20.8M	-	944.2M	16.2M	6.9M	166.8M	55.1M
	#cs-fpts	1.1M	0.4M	-	19.6M	0.7M	0.3M	21.7M	14.0M
	#cs-calls	20.0M	8.5M	-	332.8M	2.5M	1.4M	26.7M	10.1M
	Total	78.0M	29.7M	-	1296.6M	19.5M	8.6M	215.3M	79.1M
eclipse	#cs-gpts	-	7.8K	-	-	21.9K	8.0K	-	-
	#cs-pts	-	585.7M	-	-	601.8M	512.5M	-	-
	#cs-fpts	-	12.8M	-	-	16.7M	13.5M	-	-
	#cs-calls	-	345.2M	-	-	161.2M	147.3M	-	-
	Total	-	943.7M	-	-	779.7M	673.4M	-	-
fop	#cs-gpts	2.9K	1.8K	4.3K	2.0K	3.4K	1.9K	9.1K	2.2K
	#cs-pts	4.1M	1.2M	67.8M	27.5M	3.7M	1.1M	47.0M	7.9M
	#cs-fpts	0.2M	71.4K	8.0M	5.8M	0.2M	76.4K	8.2M	6.2M
	#cs-calls	1.3M	0.5M	31.0M	20.0M	0.5M	0.2M	5.1M	0.7M
	Total	5.6M	1.8M	106.7M	53.2M	4.4M	1.4M	60.4M	14.8M
luindex	#cs-gpts	2.8K	1.6K	4.5K	2.0K	3.7K	1.8K	10.6K	2.2K
	#cs-pts	4.4M	1.4M	72.6M	31.4M	4.1M	1.2M	53.0M	8.5M
	#cs-fpts	0.2M	73.0K	9.0M	6.6M	0.2M	77.3K	9.0M	6.6M
	#cs-calls	1.4M	0.6M	34.1M	22.9M	0.6M	0.3M	5.6M	0.8M
	Total	6.0M	2.0M	115.6M	60.9M	4.9M	1.6M	67.7M	15.9M
lusearch	#cs-gpts	2.9K	1.6K	4.2K	1.8K	3.7K	1.8K	10.3K	2.1K
	#cs-pts	6.8M	1.6M	193.6M	37.8M	5.4M	1.4M	116.5M	10.1M
	#cs-fpts	0.2M	77.4K	11.0M	7.9M	0.2M	82.7K	10.3M	7.9M
	#cs-calls	3.1M	0.7M	149.3M	27.8M	1.1M	0.3M	41.8M	1.0M
	Total	10.1M	2.4M	353.9M	73.6M	6.7M	1.8M	168.6M	19.0M
pmd	#cs-gpts	3.4K	1.9K	5.1K	2.1K	5.3K	2.1K	21.3K	2.4K
	#cs-pts	12.7M	5.1M	142.9M	42.0M	14.9M	4.8M	171.1M	14.8M
	#cs-fpts	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	2.2M	1.0M	17.4M	1.9M
	Total	17.2M	7.3M	212.8M	79.6M	18.2M	6.2M	205.5M	25.7M
xalan	#cs-gpts	4.9K	2.9K	-	3.2K	4.2K	2.8K	10.0K	3.2K
	#cs-pts	160.4M	49.0M	-	161.0M	51.1M	41.5M	517.5M	123.6M
	#cs-fpts	6.3M	4.3M	-	15.7M	5.4M	4.5M	33.1M	16.0M
	#cs-calls	49.6M	21.6M	-	103.4M	14.6M	13.7M	86.0M	52.8M
	Total	216.3M	74.9M	-	280.2M	71.2M	59.7M	636.6M	192.3M
checkstyle	#cs-gpts	7.7K	3.5K	-	-	10.8K	4.3K	-	5.2K
	#cs-pts	166.2M	44.7M	-	-	130.8M	38.3M	-	353.9M
	#cs-fpts	1.5M	0.4M	-	-	2.8M	0.6M	-	141.6M
	#cs-calls	86.5M	23.2M	-	-	24.1M	9.0M	-	79.8M
	Total	254.2M	68.3M	-	-	157.6M	47.9M	-	575.3M
JPC	#cs-gpts	7.3K	4.1K	21.3K	5.7K	7.0K	3.8K	16.4K	4.3K
	#cs-pts	27.8M	11.9M	606.3M	48.0M	13.2M	7.0M	67.3M	12.2M
	#cs-fpts	0.9M	0.3M	19.3M	7.2M	0.8M	0.3M	11.2M	6.7M
	#cs-calls	9.8M	5.5M	93.8M	28.8M	2.8M	2.0M	8.2M	2.0M
	Total	38.5M	17.7M	719.5M	84.1M	16.8M	9.4M	86.7M	20.8M
findbugs	#cs-gpts	34.1K	4.5K	-	6.0K	11.0K	4.5K	43.8K	5.9K
	#cs-pts	358.2M	41.2M	-	126.9M	58.6M	19.5M	553.2M	38.6M
	#cs-fpts	18.0M	1.0M	-	23.1M	5.0M	1.0M	61.1M	23.9M
	#cs-calls	147.2M	13.3M	-	84.9M	13.2M	5.8M	101.5M	5.9M
	Total	523.5M	55.5M	-	234.8M	76.8M	26.2M	715.9M	68.4M

context-sensitivity selections to be pre-configured). In this paper, however, EkOBJ underperforms ZkOBJ when both are evaluated also in SOOT (but by using a more precise on-the-fly exception analysis [34] and turning on the manual heuristic described in Section V to allow certain objects to be identified per dynamic type and then pre-configured to be always analyzed context-insensitively). As described in Section II-C, whichever is faster in whichever setting is irrelevant to this work, CONCH can boost their performance regardless (as evaluated in Section V). In the absence of this manual heuristic, CONCH is observed to be even substantially more effective in boosting the performance of an object-sensitive analysis A, since it can help A identify more context-independent objects that would otherwise be analyzed context-sensitively by A. In our evaluation, turning this manual heuristic on aims to challenge CONCH to demonstrate its performance benefits over a faster baseline, by debloating the contexts for the faster baseline (that has already been debloated manually before).

Recently, TURNER [26] exploits object containment to predict context-independent objects. In contrast, CONCH is a more principled approach and could find more context-independent objects than TURNER.

MAHJONG [12] 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.

In [13], context transformations are introduced as an alternative context abstraction to context strings (as used in kOBJ), but the practical benefits are shown to be small.

Data-driven approaches [14], [21], [39] apply machine learning to obtain various heuristics for supporting selective context-sensitivity. SCALER [40] trades precision for scalability by selecting a suitable context-sensitivity variant for each method so that the amount of points-to information is bounded.

Elsewhere [30], [41], [42], efforts have been made to improve the precision of object-sensitive pointer analysis. This thread of research is orthogonal to ours considered here.

Finally, unlike whole-program analyses [1], [11], [20], [36], [43], [44] considered in this paper, demand-driven pointer analyses [45]–[50] typically only compute the points-to information for program points that may affect a particular site of interest for specific clients.

VII. CONCLUSION AND FUTURE WORK

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 paper, 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) by a set of three necessary conditions that can be linearly verifiable in terms of the number of statements in the program for determining the context-dependability of any object. Our evaluation shows that our new approach, CONCH, can improve significantly the efficiency and scalability of not only kOBJ but also existing approaches to selective context-sensitivity that can already accelerate the performance of kOBJ.

We believe that the performance benefits of context debloating are not just limited to object-sensitive pointer analysis as demonstrated here. In our future work, we plan to explore how to apply context debloating to other flavors of pointer analysis such as call-site-sensitive pointer analysis [51] and context-transformation-based pointer analysis [13]. In addition, we will also investigate how to apply context debloating to other context-sensitive program analyses such as taint analysis [8] and data-dependence analysis [52] for improving their efficiency and scalability, particularly for large codebases.

VIII. ACKNOWLEDGEMENT

Thanks to all the reviewers for their constructive comments. This research is supported by an ARC Grant DP180104069.

REFERENCES

[1] O. Lhoták and L. Hendren, "Scaling Java points-to analysis using Spark," in *International Conference on Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 153–169.

[2] K. Ali and O. Lhoták, "Application-only call graph construction," in *ECOOP 2012 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 688–712.

[3] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[4] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2007, pp. 112–122.

[5] Y. Li, T. Tan, Y. Zhang, and J. Xue, "Program tailoring: Slicing by sequential criteria," in *30th European Conference on Object-Oriented Programming*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 15:1–15:27.

[6] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2006, pp. 308–319.

[7] Y. Liu and A. Milanova, "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*, ser. PASTE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 50–56.

[8] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269.

[9] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in Android apps," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 167–177.

[10] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue, "Performance-boosting sparsification of the ifds algorithm with applications to taint analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. San Diego, CA, USA: IEEE, 2019, pp. 267–279.

[11] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: understanding object-sensitivity," in *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 17–30.

[12] T. Tan, Y. Li and J. Xue, "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*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 278–291.

[13] R. Thiessen and O. Lhoták, "Context transformations for pointer analysis," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2017, p. 263–277.

[14] M. Jeon, S. Jeong, and H. Oh, "Precise and scalable points-to analysis via data-driven context tunneling," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[15] J. Lu and J. Xue, "Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. USA: IBM Corp., 2010, pp. 214–224.

[17] Y. Smaragdakis, "Doop-framework for Java pointer and taint analysis (using P/Taint)," 2021. [Online]. Available: https://bitbucket.org/yanniss/doop/

[18] IBM, "WALA: T.J. Watson Libraries for Analysis," 2020. [Online]. Available: http://wala.sourceforge.net/

[19] A. Milanova, A. Rountev, and B. G. Ryder, "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*. New York, NY, USA: Association for Computing Machinery, 2002, pp. 1–11.

[20] ——, "Parameterized object sensitivity for points-to analysis for Java," *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 1, pp. 1–41, 2005.

[21] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven context-sensitivity for points-to analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 100, 2017.

[22] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[23] B. Hassanshahi, R. K. Ramesh, P. Krishnan, B. Scholz, and Y. Lu, "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*. New York, NY, USA: Association for Computing Machinery, 2017, p. 13–18.

[24] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2014, pp. 485–495.

[25] J. Lu, D. He, and J. Xue, "Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity," *ACM Transactions on Software Engineering and Methodology*, 2021.

[26] D. He, J. Lu, Y. Gao, and J. Xue, "Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability," in *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 194. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 16:1–16:31.

[27] T. Reps, "Undecidability of context-sensitive data-dependence analysis," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 1, pp. 162–186, 2000.

[28] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 49–61.

[29] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[30] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," in *International Static Analysis Symposium*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 489–510.

[31] T. Reps, "Program analysis via graph reachability," *Information and software technology*, vol. 40, no. 11-12, pp. 701–726, 1998.

[32] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, no. 1, p. 1–69, 2015.

[33] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of Android applications in DroidSafe," in *NDSS*, vol. 15. The Internet Society, 2015, p. 110.

[34] M. Bravenboer and Y. Smaragdakis, "Exception analysis and points-to analysis: Better together," in *Proceedings of the 18th International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2009, p. 1–12.

[35] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proceedings of the 33rd International Conference on Software Engineering*. Honolulu, HI, USA: IEEE, 2011, pp. 241–250.

[36] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 243–262.

[37] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "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*. New York, NY, USA: Association for Computing Machinery, 2006, pp. 169–190.

[38] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Transactions on Programming Languages and Systems*, vol. 42, no. TOPLAS, pp. 1–40, 2020.

[39] M. Jeon, M. Lee, and H. Oh, "Learning graph-based heuristics for pointer analysis without handcrafting application-specific features," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[40] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "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*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 129–140.

[41] A. Milanova, "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*. New York, NY, USA: Association for Computing Machinery, 2007, p. 25–30.

[42] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2013, p. 423–434.

[43] J. Whaley and M. S. Lam, "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*. New York, NY, USA: Association for Computing Machinery, 2004, pp. 131–144.

[44] L. Li, C. Cifuentes, and N. Keynes, "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*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 343–353.

[45] M. Sridharan, D. Gopan, L. Shan, and R. Bodík, "Demand-driven points-to analysis for Java," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: Association for Computing Machinery, 2005, p. 59–76.

[46] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for Java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2006, p. 387–400.

[47] D. Yan, G. Xu, and A. Rountev, "Demand-driven context-sensitive alias analysis for Java," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2011, pp. 155–165.

[48] L. Shang, X. Xie, and J. Xue, "On-demand dynamic summary-based points-to analysis," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 264–274.

[49] Y. Sui and J. Xue, "On-demand strong update analysis via value-flow refinement," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 460–473.

[50] J. Späth, L. Nguyen Quang Do, K. Ali, and E. Bodden, "Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java," in *30th European Conference on Object-Oriented Programming*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 22:1–22:26.

[51] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences , 1978.

[52] Q. Zhang and Z. Su, "Context-sensitive data-dependence analysis via linear conjunctive language reachability," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 344–358.