Accelerating Object-Sensitive Pointer Analysis by Exploiting Object Containment and Reachability

Dongjie He
UNSW Sydney, Australia

Jingbo Lu
UNSW Sydney, Australia

Yaoqing Gao
Huawei, Canada

Jingling Xue
UNSW Sydney, Australia

Abstract

Object-sensitive pointer analysis for an object-oriented program can be accelerated if context-sensitivity can be selectively applied to some precision-critical variables/objects in the program. Existing pre-analyses, which are performed to make such selections, either preserve precision but achieve limited speedups by reasoning about all the possible value flows in the program conservatively or achieve greater speedups but sacrifice precision (often unduly) by examining only some but not all the value flows in the program heuristically. In this paper, we introduce a new approach, named TURNER, that represents a sweet spot between the two existing ones, as it is designed to enable object-sensitive pointer analysis to run significantly faster than the former approach and achieve significantly better precision than the latter approach. TURNER is simple, lightweight yet effective due to two novel aspects in its design. First, we exploit a key observation that some precision-uncritical objects can be approximated based on the object-containment relationship pre-established (by applying Andersen's analysis). This approximation introduces a small degree yet the only source of imprecision into TURNER. Second, leveraging this initial approximation, we introduce a simple DFA to reason about object reachability for a method intra-procedurally from its entry to its exit along all the possible value flows established by its statements to finalize its precision-critical variables/objects identified. We have validated TURNER with an implementation in SOOT against the state of the art using a set of 12 popular Java benchmarks and applications.

2012 ACM Subject Classification Theory of Computation → Program Analysis

Keywords and phrases Object-Sensitive Pointer Analysis, CFL Reachability, Object Containment

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.18

Acknowledgements We thank the reviewers for their constructive comments. This work is supported by an ARC DP grant (DP180104069) and a UNSW-Huawei research grant (YBN2019105002).

1 Introduction

Pointer analysis is a significant static program analysis that approximates the potential runtime values (memory locations) for the pointer variables in a program. It plays an important role in a wide range of real-world applications, including security analysis [2, 10], program verification [8], program understanding [36, 20], and bug detection [25, 11].

For object-oriented languages like Java, *context sensitivity*, which distinguishes the variables declared and objects allocated locally in a method under different calling contexts, is widely enforced in developing highly precise pointer analyses. In general, a context is represented by a sequence of k context elements (under k limiting). There are two common forms of context-sensitivity: k-call-site-sensitivity [29] (which distinguishes the contexts of a method by its k-most-recent call sites) and k-object-sensitivity [23] (which distinguishes the contexts of a method by its receiver object's k-most-recent allocation sites). The latter is widely regarded as a better abstraction in achieving precision and efficiency [31, 39, 41, 12, 22].

However, k-object-sensitive pointer analysis (with k-object-sensitivity as its context abstraction), denoted kOBJ, still does not scale well for reasonably large programs when $k \geqslant 3$ and is often time-consuming when it is scalable [31, 39, 41, 12]. As k increases, blindly applying a k-limiting context abstraction uniformly to a program can cause the number of contexts handled to blow up exponentially (often without improving precision much).

In this paper, we address the problem of developing a pre-analysis for a Java program to enable kOBJ to apply context-sensitivity (i.e, a k-limited context abstraction) only to some of its variables/objects selected and context-insensitivity to all the rest in the program.

▶ **Definition 1.** *A variable/object n in a program is* precision-critical *if kOBJ loses precision in terms of the points-to information obtained (for some value of k) when n is analyzed by kOBJ context-insensitively instead of context-sensitively.*

A pre-analysis is said to be *precision-preserving* if it can identify the precision-critical variables/objects in a program precisely or over-approximately as being context-sensitive, and *non-precision-preserving* otherwise. Unfortunately, making such selections precisely is out of question as solving kOBJ without k-limiting is undecidable [27]. When designing a practical pre-analysis, which aims to select the set of context-sensitive variables/objects, C_{ideal}, in the program, the main challenge are to ensure that (1) C_{ideal} includes as many precision-critical variables/objects as possible but as few precision-uncritical variables/objects as possible, (2) C_{ideal} results in no or little precision loss, and (3) C_{ideal} is found in a lightweight manner to ensure that the pre-analysis overhead introduced is negligible (relative to kOBJ).

Recently, several pre-analyses have been proposed [32, 13, 9, 19, 22, 21]. Broadly speaking, two approaches exist. EAGLE [22, 21] represents a precision-preserving acceleration of kOBJ by reasoning about CFL (Context-Free-Language) reachability in the program. Designed to be precision-preserving, EAGLE analyzes conservatively and often efficiently the value flows reaching a variable/object and selects the set of context-sensitive variables/objects as a superset of the set of precision-critical variables/objects in the program over-approximately, thereby limiting the potential speedups achieved. On the other hand, ZIPPER [19], as a non-precision-preserving representative of the remaining pre-analyses [32, 13, 9, 19], examines the value flows reaching a variable/object heuristically and often efficiently by selecting the set of context-sensitive variables/objects to include some but not all the precision-critical variables/objects and also some precision-uncritical variables/objects in the program. As a result, ZIPPER can sometimes improve the efficiency of kOBJ more significantly than EAGLE but at the expense of introducing a substantial loss of precision for some programs.

In this paper, we introduce a new approach, named TURNER, that represents a sweet spot between EAGLE and ZIPPER: TURNER enables kOBJ to run significantly faster than EAGLE while achieving significantly better precision than ZIPPER. Despite losing a small precision in the average points-to set size (#avg-pts), TURNER achieves exactly the same precision for the other three commonly used precision metrics [31, 39, 41, 12, 22, 21], call graph construction (#call-edges), may-fail casting (#may-fail-casts) and polymorphic call detection (#poly-calls), for a set of 12 popular Java benchmarks and applications evaluated. TURNER is simple, lightweight yet effective due to two novel aspects in its design. First, we exploit a key observation that some precision-uncritical objects can be approximated initially based on the object-containment relationship that is inferred from the points-to information pre-computed by Andersen's analysis [1]. This approximation turns out to be practically accurate, as it introduces a small degree yet the only source of imprecision into the final points-to information computed. Second, leveraging this initial approximation, we introduce a simple DFA (Deterministic Finite Automaton) to reason about object reachability across a method (from its entry to its exit) intra-procedurally along all the possible value flows established by its statements to finalize all its precision-critical variables/objects selected.

We have validated TURNER with an implementation in SOOT against EAGLE and ZIPPER using a set of 12 Java benchmarks and applications. In general, TURNER enables kOBJ to run significantly faster than EAGLE due to fewer precision-uncritical variables/objects analyzed context-sensitively and achieve significantly better precision than ZIPPER due to more precision-critical variables/objects analyzed context-sensitively than ZIPPER.

In summary, our paper makes the following contributions:

- We introduce a new approach, TURNER, that can accelerate k-object-sensitive pointer analysis (i.e., kOBJ) for Java programs significantly with negligible precision loss.
- We propose to first approximate the precision-criticality of the objects in a program based on object containment and then decide whether the variables/objects in the program should be context-sensitive or not by conducting an object reachability analysis intra-procedurally with a DFA, which turns out to be simple, lightweight and effective.
- TURNER enables kOBJ to run significantly faster than EAGLE and achieve significantly better precision than ZIPPER for a set of 12 popular Java benchmarks and applications evaluated in terms of four common precision metrics, #avg-pts, #call-edges, #may-fail-casts, and #poly-calls (with TURNER losing no precision for the last three metrics).

The rest of this paper is organized as follows. Section 2 motivates our TURNER approach. Section 3 gives a version of kOBJ that supports selective context-sensitivity. Section 4 formalizes our TURNER approach. In Section 5, we evaluate TURNER against the state of the art. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 Motivation

We motivate TURNER in the context of the two state-of-the-art pre-analyses, EAGLE [22, 21] and ZIPPER [19]. EAGLE supports partial context-sensitivity as it enables kOBJ to analyze only a subset of variables/objects in a method context-sensitively. On the other hand, ZIPPER allows kOBJ to analyze a method either fully context-sensitively or fully context-insensitively. Like EAGLE, TURNER also supports partial context-sensitivity in order to maximize the potential speedups attainable. As in EAGLE and ZIPPER, TURNER also relies on the points-to information in a program pre-computed by Andersen's analysis [1] (context-insensitively).

In Section 2.1, we give some background information. In Section 2.2, we examine the main challenges faced in developing a pre-analysis for accelerating kOBJ and discuss the methodological differences between TURNER and two existing approaches, EAGLE and ZIPPER. In Section 2.3, we introduce a motivating example abstracted from real code by highlighting the effects of these differences on the context-sensitivity selectively applied to kOBJ. In Section 2.4, we describe the basic idea behind TURNER (including our insights and trade-offs).

2.1 Background

In object-sensitive pointer analysis [23], the calling contexts of a method are distinguished by its receiver objects. Let each allocation site be abstracted by one unique object. In kOBJ, an object o_1 is modeled context-sensitively by a *heap context* of length $k-1$, $[o_2, ..., o_k]$, where o_i is the receiver object of a method in which o_{i-1} is allocated. As a result, a method with o_1 as its receiver object will be analyzed context-sensitively multiple times, once for each of

o_1's heap contexts $[o_2, ..., o_k]$, i.e., once under every possible *method context* $[o_1, ..., o_k]$ of length k. Thus, kOBJ can be specified by either heap or method contexts alone.

Given a variable v analyzed under a method context c, its context-sensitive points-to set is expressed as $\mathsf{pts}(v, c) = \{(o_1, c_1), \cdots, (o_n, c_n)\}$, where each pointed-to object o_i is identified by its heap context c_i. Let M_v be the set of method contexts under which v is analyzed. Then the context-insensitive points-to set for v can be found as $\overline{\mathsf{pts}}(v) = \bigcup_{c \in M_v} \{o \mid (o, c') \in \mathsf{pts}(v, c)\}$. When comparing different context-sensitive pointer analyses precision-wise, the context-insensitive points-to information thus obtained is used, as is done in the literature [32, 12, 13, 39, 19, 21].

2.2 Challenges

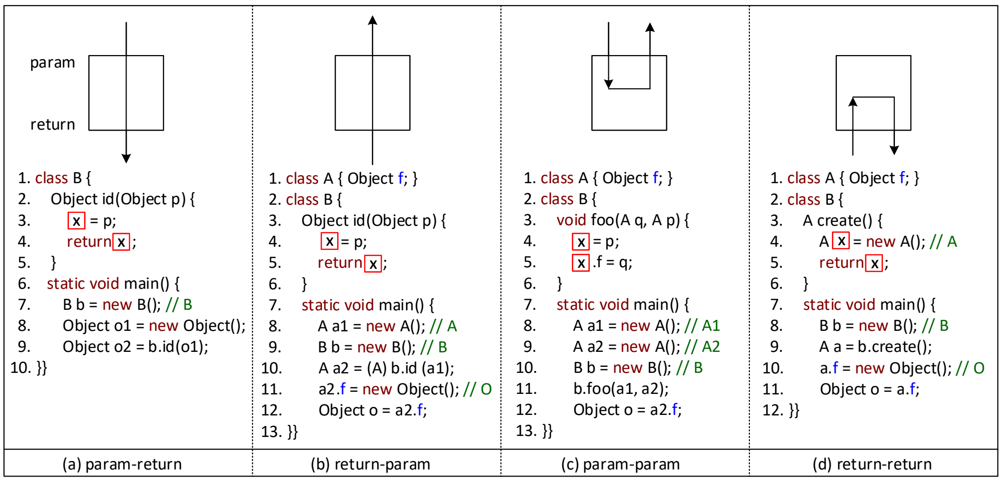
A variable/object n in a program is precision-critical if kOBJ loses precision when it analyzes n context-insensitively instead of context-sensitively (Definition 1). In the case of a precision loss, there must exist some variable v in the program such that its context-insensitive points-to information becomes less precise. In this case, $\overline{\mathsf{pts}}(v)$ will contain not only the pointed-to objects found before (when n is analyzed context-sensitively) but also some spurious pointed-to objects introduced now (when n is analyzed context-insensitively). As n and v can be further apart in the program, separated by a long sequence of method calls (with complex field accesses on n along the way), designing a practical pre-analysis P, which selects a set of variables/objects in a program for kOBJ to analyze context-sensitively, is challenging (since solving kOBJ without k-limiting is undecidable [27]). For a program, let C_{ideal} be the set of precision-critical variables/objects specified by Definition 1 and C_P the set of context-sensitive variables/objects selected by P. The main challenges lie in how to ensure that (1) $|C_{\text{ideal}} - C_P|$ is minimized (so that as many precision-critical variables/objects are selected) and $|C_P - C_{\text{ideal}}|$ is minimized (so that as few precision-uncritical variables/objects are selected), (2) C_P causes kOBJ to lose no or little precision, and (3) C_P is selected in a lightweight manner (so that P introduces negligible overhead relative to kOBJ).

A pre-analysis for kOBJ relies on the following fact to identify a precision-critical variable/object, with its accesses possibly triggered by statements outside its containing method. Without loss of generality, a method is assumed to contain only one return statement of the form "**return** r", where r a local variable in the method (referred to as its *return variable*).

▶ **Fact 2.** *Consider a program being analyzed object-sensitively with the parameters and the return variable of a method modeled as its (special) fields as in [22, 21]. A variable/object n in a method M in the program is considered to be precision-critical* only if*, during program execution, there is a value flow entering and leaving M via a parameter or the return variable of M, by passing through n (i.e., by first writing into n via an access path and then reading it from the same access path), where n may be the parameter or the return variable itself.*

In this case, analyzing n context-sensitively will allow several such value flows to be tracked separately based on their calling contexts. Otherwise, some precision may be potentially lost.

A pre-analysis, as illustrated in Figure 1, should identify a (local) variable `x` as precision-critical by considering a total of four possible value-flow patterns passing through `x` (classified according to whether the two end points of a value-flow are a parameter or the return variable of its containing method [34, 22]). The same four patterns are also applicable to a locally allocated object. In "*param-return*" (Figure 1(a)), the pre-analysis should recognize that the object created in line 8 will flow into `x` in `id()` via its parameter `p` and then out of `id()` via a return variable, which happens to be `x` itself. In "*return-param*" (Figure 1(b)), the pre-analysis, when checking whether the object created in line 11 will flow into `o` in line 12 or



```
1. class B {
2.   Object id(Object p) {
3.     x = p;
4.     return x;
5.   }
6.   static void main() {
7.     B b = new B(); // B
8.     Object o1 = new Object();
9.     Object o2 = b.id(o1);
10. }}
```

(a) param-return

```
1. class A { Object f; }
2. class B {
3.   Object id(Object p) {
4.     x = p;
5.     return x;
6.   }
7.   static void main() {
8.     A a1 = new A(); // A
9.     B b = new B(); // B
10.    A a2 = (A) b.id (a1);
11.    a2.f = new Object(); // O
12.    Object o = a2.f;
13. }}
```

(b) return-param

```
1. class A { Object f; }
2. class B {
3.   void foo(A q, A p) {
4.     x = p;
5.     x.f = q;
6.   }
7.   static void main() {
8.     A a1 = new A(); // A1
9.     A a2 = new A(); // A2
10.    B b = new B(); // B
11.    b.foo(a1, a2);
12.    Object o = a2.f;
13. }}
```

(c) param-param

```
1. class A { Object f; }
2. class B {
3.   A create() {
4.     A x = new A(); // A
5.     return x;
6.   }
7.   static void main() {
8.     B b = new B(); // B
9.     A a = b.create();
10.    a.f = new Object(); // O
11.    Object o = a.f;
12. }}
```

(d) return-return

Figure 1 A total of four possible value-flow patterns for determinng whether a variable `x` should be precision-critical or not.

not, will first need to find out what `a2` points to. This will entail reasoning about the value flow of `a2` in reverse order, by entering `id()` via its return statement (variable) and leaving `id()` from its parameter `p`. In "*param-param*" (Figure 1(c)), the object `A1` created in line 8 will flow into `x` (or `x.f` precisely) in `foo()` via its parameter `q` and then out of `foo()` via its parameter `p`. In "*return-return*" (Figure 1(d)), the pre-analysis, when checking whether the object created in line 10 can flow into `o` in line 11 or not, will need to find what `a` points to, by entering and exiting `create()` from its return variable and visiting `x` in between.

We can now discuss how TURNER differs from EAGLE [22, 21] and ZIPPER [19] methodologically. To start with, all the three are relatively lightweight with respect to kOBJ. Below we examine these pre-analyses in terms of their efficiency and precision tradeoffs made on approximating C_{ideal}. There are two caveats. First, C_{ideal} is conceptual but cannot be found exactly in a program. Second, some precision-critical variables/objects affect the precision and/or efficiency of kOBJ more profoundly than others, but they cannot be easily identified. How to do so approximately can be an interesting research topic in future work.

EAGLE is precision-preserving, since it accounts for all the four value-flow patterns given in Figure 1 by reasoning about CFL reachability in the program inter-procedurally to ensure that $C_{\text{ideal}} - C_{\text{EAGLE}} = \varnothing$. For some programs, EAGLE may conservatively misclassify many precision-uncritical variables/objects as being precision-critical, thereby causing $C_{\text{EAGLE}} - C_{\text{ideal}}$ to be unduly large, and consequently, limiting the speedups attainable.

ZIPPER is not precision-preserving (implying that $C_{\text{ideal}} - C_{\text{ZIPPER}} \neq \varnothing$, in general), since it considers only the "*param-return*" and "*return-param*" patterns in Figure 1 heuristically by pattern-matching and ignores "*param-param*" (according to its authors [19]) and "*return-return*" (according to its open-source implementation). For some programs, ZIPPER can achieve greater speedups than EAGLE (under certain configurations that dictate how certain objects should be analyzed) but at a precision loss, since it has misclassified some precision-yet performance-critical variables/objects as context-insensitive.

In this paper, TURNER is designed to strike a good balance between EAGLE and ZIPPER. We aim to ensure that $|C_{\text{TURNER}} - C_{\text{ideal}}| < |C_{\text{EAGLE}} - C_{\text{ideal}}|$ so that TURNER can enable kOBJ to run significantly faster than EAGLE (due to fewer precision-uncritical variable/objects selected for kOBJ to analyze context-sensitively). At the same time, we aim to ensure that $|C_{\text{ideal}} - C_{\text{TURNER}}| < |C_{\text{ideal}} - C_{\text{ZIPPER}}|$ so that TURNER can also enable kOBJ to achieve significantly

```
 1. class Entry {
 2.   Object key, value;
 3.   Entry(Object p, Object q) {
 4.     this.key = p;
 5.     this.value = q;
 6. }}

 7. class HashMap {
 8.   Entry[] table;
 9.   Object get(Object k){
10.     int idx = k.hashCode;
11.     Entry[] t = this.table;
12.     Entry e = t[idx];
13.     Object r = e.value;
14.     return r;
15.   }
16.   void put(Object k, Object v) {
17.     int idx = k.hashCode;
18.     Entry e = new Entry(k, v);  // E
19.     Entry[] t = this.table;
20.     t[idx] = e;
21.   }
22.   HashMap() {
23.     Entry[] t = new Entry[16]; // @
24.     this.table = t;
25. }}

26. class A {
27.   void foo(Object k) {
28.     HashMap map1 = new HashMap(); // M1
29.     HashMap map2 = new HashMap(); // M2
30.     Object v1 = new Object(); // O1
31.     Object v2 = new Object(); // O2
32.     map1.put(k, v1);
33.     map2.put(k, v2);
34.     Object w1 = map1.get(k);
35.     Object w2 = map2.get(k);
36.   }
37.   public static void main(String args[]) {
38.     Object k = new Object(); // O
39.     A ai = new A(); // Ai  ┐
40.     ai.foo(k);             ├1 ≤ i ≤ n
41.     ...                    ┘
42. }}
```

Figure 2 A Java program abstracted from real code using the standard JDK library.

better precision than Zipper (due to more precision-critical variable/objects selected for $k$obj to analyze context-sensitively). We accomplish this by exploiting object containment to approximate the precision-criticality of objects and then reasoning about object reachability by considering all the four value-flow patterns in Figure 1 intra-procedurally.

2.3 Example

Figure 2 gives a Java program abstracted from real code developed based on JDK. In lines 1-25, a simplified `HashMap` class is defined. In lines 26-42, class `A` represents a use case of `HashMap`. In `foo()`, two instances of `HashMap`, `M1` and `M2`, and two instances of `java.lang.Object`, `O1` and `O2`, are created. Afterwards, `O1` (`O2`), pointed to by `v1` (`v2`), is deposited into `M1` (`M2`), pointed to by `map1` (`map2`), with `O` (received from its parameter `k`) as the corresponding key, and later retrieved and saved in `w1` (`w2`). In `main()`, n instances of `A`, $\texttt{A}_1, \ldots, \texttt{A}_n$, are created (where $n > 1$) and then used as the receivers for invoking `foo()`.

Table 1 lists the contexts used for analyzing this program by the four main analyses, 2obj, E-2obj, Z-2obj, and T-2obj. Here, P-2obj denotes the version of 2obj that adopts the selective context-sensitivity prescribed by $P \in \{$E (for Eagle), Z (for Zipper), T (for Turner)$\}$. Eagle is always precision-preserving. For this program, Zipper happens to be also precision-preserving since Z-2obj behaves exactly as 2obj does. Turner also happens to be precision-preserving but T-2obj differs from 2obj/Z-2obj and E-2obj substantially. Below we focus on examining how the context-insensitive points-to information for `w1` and `w2` in `foo()`, $\overline{\mathsf{pts}}(\texttt{w1}) = \{\texttt{O1}\}$ and $\overline{\mathsf{pts}}(\texttt{w2}) = \{\texttt{O2}\}$, is obtained by each of the four main analyses. For reasons of symmetry, Figure 3 illustrates only how $\overline{\mathsf{pts}}(\texttt{w1}) = \{\texttt{O1}\}$ is obtained.

First of all, 2obj analyzes `foo()` for a total of n times by identifying its variables/objects under the i-th invocation with its receiver $\texttt{A}_i$ (Figure 3(a)). Thus, $\forall\ 1 \leqslant i \leqslant n : \mathsf{pts}(\texttt{w1}, [\texttt{A}_i]) = \{\texttt{O1}, [\texttt{A}_i]\} \wedge \mathsf{pts}(\texttt{w2}, [\texttt{A}_i]) = \{\texttt{O2}, [\texttt{A}_i]\}$ context-sensitively. By projecting out all the contexts,

Table 1 The contexts used for analyzing the variables/objects in Figure 2 by 2OBJ, E-2OBJ, Z-2OBJ, and T-2OBJ (where i in each context containing $\mathtt{A}_i/\mathtt{a}_i$ ranges over $[1, n]$).

Method	Variables/Objects	2OBJ/ Z-2OBJ	E-2OBJ	T-2OBJ
Entry	p, q, this	[E, M1], [E, M2]	[E, M1], [E, M2]	[E, M1], [E, M2]
get	k	[M1, A_i], [M2, A_i]	[]	[]
	e, r, this, t		[M1, A_i], [M2, A_i]	[M1], [M2]
put	k, v, e, this, t	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1], [M2]
	E	[M1], [M2]	[M1], [M2]	
HashMap	this, t	[M1, A_i], [M2, A_i]	[M1, A_i], [M2, A_i]	[M1], [M2]
	@	[M1], [M2]	[M1], [M2]	
foo	v1, v2, w1, w2	[A_i]	[]	[]
	O1, O2			
	k, map1, map2		[A_i]	
	M1, M2			
main	k, a_i	[]	[]	[]
	O, A_i			

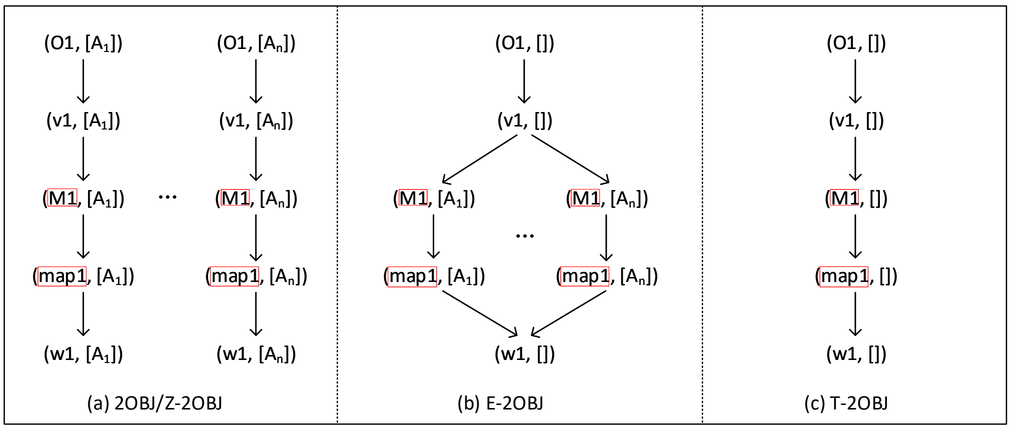

(O1, [A1]) → (v1, [A1]) → (M1, [A1]) → (map1, [A1]) → (w1, [A1])
(On, [An]) … (O1, [An]) → (v1, [An]) → (M1, [An]) → (map1, [An]) → (w1, [An])
(a) 2OBJ/Z-2OBJ
(O1, []) → (v1, []) → (M1, [A1]) … (M1, [An]) → (map1, [A1]) … (map1, [An]) → (w1, [])
(b) E-2OBJ
(O1, []) → (v1, []) → (M1, []) → (map1, []) → (w1, [])
(c) T-2OBJ

Figure 3 Computing $\overline{\mathtt{pts}}(\mathtt{w1}) = \{\mathtt{O1}\}$ for Figure 2 by 2OBJ, E-2OBJ, Z-2OBJ and T-2OBJ.

2OBJ obtains $\overline{\mathtt{pts}}(\mathtt{w1}) = \{\mathtt{O1}\}$ and $\overline{\mathtt{pts}}(\mathtt{w2}) = \{\mathtt{O2}\}$ context-insensitively, as desired.

For this particular program, Z-2OBJ is equivalent to 2OBJ (Table 1 and Figure 3(a)). However, it is easy to modify it slightly so that Z-2OBJ will behave differently while suffering from a loss of precision (as it does not consider the last two patterns given in Figure 1).

E-2OBJ enables 2OBJ to support partial context-sensitivity without losing any precision. The variables/objects in $\{\mathtt{v1}, \mathtt{v2}, \mathtt{w1}, \mathtt{w2}, \mathtt{O1}, \mathtt{O2}\}$ in `foo()` and variable `k` in `get()` will now be context-insensitive. In the case of `foo()`, however, `k`, `map1`, `map2`, `M1` and `M2` must still be analyzed context-sensitively due to a spurious "*param-return*" pattern established by the facts that (1) `k` is a parameter, (2) `put()` can write into `M1`/`M2`, and (3) `get()` can read from `M1`/`M2`. As a result, as illustrated in Figure 3(b), E-2OBJ will still need to analyze `foo()` for a total of n times, since it must distinguish the two `HashMap` objects `M1` and `M2` created in `foo()` context-sensitively as in 2OBJ, except that it can now analyze the two objects, `O1` and `O2`, created in `foo()` context-insensitively. Thus, E-2OBJ obtains directly that $\mathtt{pts}(\mathtt{w1}, [\,]) = \{\mathtt{O1}, [\,]\}$ and $\mathtt{pts}(\mathtt{w2}, [\,]) = \{\mathtt{O2}, [\,]\}$, i.e., $\overline{\mathtt{pts}}(\mathtt{w1}) = \{\mathtt{O1}\}$ and $\overline{\mathtt{pts}}(\mathtt{w2}) = \{\mathtt{O2}\}$.

T-2OBJ, as illustrated in Figure 3(c), goes beyond E-2OBJ (for this particular program) by modeling `M1` and `M2` also context-insensitively. As a result, `foo()` is analyzed context-insensitively only once. As in the case of E-2OBJ, T-2OBJ also obtains directly that $\mathtt{pts}(\mathtt{w1}, [\,]) = \{\mathtt{O1}, [\,]\}$ and $\mathtt{pts}(\mathtt{w2}, [\,]) = \{\mathtt{O2}, [\,]\}$, i.e., $\overline{\mathtt{pts}}(\mathtt{w1}) = \{\mathtt{O1}\}$ and $\overline{\mathtt{pts}}(\mathtt{w2}) = \{\mathtt{O2}\}$.

2.4 Our Approach

TURNER is designed to accelerate kOBJ with partial context-sensitivity at a negligible loss of precision. Unlike EAGLE [22, 21] and ZIPPER [19], TURNER works by exploiting both object containment and object reachability to enable kOBJ to strike a better balance between efficiency and precision. In principle, TURNER may lose precision in its first stage only but will always preserve precision in its second stage if it does not lose precision in its first stage.

2.4.1 Object Containment

To start with, we exploit a key insight stated below to identify some precision-uncritical objects approximately based on the object containment relationship that is inferred from the points-to information pre-computed (context-insensitively) by Anderson's analysis [1].

▶ **Observation 3.** *A* top container *is an object that is pointed to by neither (1) another object (which may be the container itself) via a field of a declared type of C or C*`[]`*, where C is a class type nor (2) the return variable of the method in which the container is allocated.*

A bottom container *is an object that does not point to another object (which may be the container itself) via a field of a declared type of C or C*`[]`*, where C is a class type.*

Given a program, its top and bottom containers are considered as being precision-uncritical.

▶ **Definition 4.** *Observation 3 is said to be precision-preserving for a program if kOBJ does not lose precision when it analyzes the precision-uncritical objects identified in the program context-insensitively and the remaining variables/objects exactly as before.*

Therefore, an object created by a factory method (regarded here as a method that returns its own allocated objects via its return variable) is not a top container. Such an object will be considered as being precision-uncritical iff it is a bottom container. For a program, the precision-uncritical objects identified here will be analyzed by kOBJ context-insensitively (for the reasons given below) and the remaining objects will be further classified as either precision-critical or precision-uncritical by an object reachability analysis (Section 2.4.2).

Consider `create()` in Figure 1(d). The object `A` created inside is not regarded as a top container, since it is pointed to by its return variable. In object-sensitive pointer analysis, when `create()` called on receiver object `B` in line 9 is analyzed, returning `A` to this caller is actually modeled as `this.ret = x` (line 5) and `a = b.ret` (line 9), where both `this` and `b` point to `B`, and `ret` can be understood as a special return variable introduced for `create()` (Section 4.2.2.2) [22, 21]. Conceptually, `A` is not a top container. In this example, `A` is not a bottom container either, since `A.f = O` in line 10, where `O` is an instance of `java.lang.Object`. As a result, `A` is considered as being precision-critical. However, if lines 10-11 were not present, then `A` would be deemed as being precision-uncritical as it is now a bottom container.

Consider Figure 2 (which is free of factory methods), where a total of $n + 7$ objects can be found: `E`, `@`, `M1`, `M2`, `O1`, `O2`, `O`, $\texttt{A}_1, ..., \texttt{A}_n$. According to the object containment relationship inferred from Andersen's analysis, `M1` and `M2` contain `@`, which contains `E`, which contains `O1`, `O2` and `O`. By Observation 3, the set of top containers is given by $\{\texttt{M1}, \texttt{M2}, \texttt{A}_1, ...\texttt{A}_n\}$ and the set of bottom containers is given by $\{\texttt{O1}, \texttt{O2}, \texttt{O}, \texttt{A}_1, ...\texttt{A}_n\}$. Note that both sets of containers are not necessarily disjoint. Thus, the $n + 5$ objects in $\{\texttt{M1}, \texttt{M2}, \texttt{O1}, \texttt{O2}, \texttt{O}, \texttt{A}_1, ...\texttt{A}_n\}$ are considered as being precision-uncritical and will thus be analyzed by kOBJ context-insensitively.

In our approach, Observation 3 (made based on object containment) represents the only source of imprecision in TURNER, which may propagate into its object reachability analysis. TURNER will suffer only a slight loss of precision in #avg-pts computed by T-kOBJ when some top or bottom containers that should be context-sensitive are mis-classified as being precision-uncritical, and consequently, analyzed by T-kOBJ context-insensitively. However, this does not affect the precision of #call-edges, #may-fail-casts, and #poly-calls for the set of 12 popular Java programs evaluated (at least). The set of top containers consists of the objects that are allocated and used locally in a method, such as `M1` and `M2` (two `HashMap` objects) in `foo()` in Figure 2. These objects do not require context-sensitivity, since their encapsulated data does not usually flow out of its containing methods via their parameters or return variables. On the other hand, a bottom container also does not usually require context-sensitivity, as it represents an object that typically encapsulates its primitive data (if any), including arrays of primitive types if it ever contains pointers, such as `O`, `O1` and `O2` (three field-less `java.lang.Object` objects) in Figure 2. In Section 5.3, we will examine two examples to explain why TURNER loses some small precision in #avg-pts but preserves precision in #call-edges, #may-fail-casts, and #poly-calls in real code.

2.4.2 Object Reachability

Given a program, TURNER relies on a simple DFA to reason about implicitly the four value-flow patterns in Figure 1 in a method to select its variables/objects to be analyzed by T-kOBJ context-sensitively. By design, the precision-uncritical objects identified by Observation 3 in the program are deemed context-insensitive. The remaining objects in the program will be classified by the DFA as either precision-critical (context-sensitive) or precision-uncritical (context-insensitive). Simultaneously, the variables in the program are classified. TURNER's intra-procedural analysis will be precision-preserving if Observation 3 is precision-preserving, as it is designed to over-approximate the precision-critical variables/objects in the program.

For our example in Figure 2, Table 1 gives the contexts selected by TURNER for kOBJ, i.e., T-2OBJ. We discuss only their differences with the contexts selected by EAGLE for kOBJ, i.e., E-2OBJ. By exploiting object containment as discussed in Section 2.4.1, `M1`, `M2`, `O1`, `O2`, and `O` have been identified as being precision-uncritical and will thus be analyzed context-insensitively. Given that `M1` and `M2`, are now context-insensitive, `k`, `map1`, and `map2` will also be identified as being context-insensitive by our DFA, as the spurious "*param-param*" pattern that causes EAGLE to flag `M1`, `M2`, `k`, `map1`, and `map2` in `foo()` as being context-sensitive no longer exists (Section 2.3). As `M1` and `M2` are context-insensitive, the contexts $[\mathtt{M1}, \mathtt{A}_i]$ and $[\mathtt{M2}, \mathtt{A}_i]$ listed under E-2OBJ have been shortened to [`M1`] and [`M2`] under T-2OBJ (Table 1).

3 Preliminaries

We take a standard formalization of kOBJ [23] from [35] and adapt it to support selective context-sensitivity. This gives a formal basis to understand our pre-analysis introduced.

3.1 A Simplified Object-Oriented Language

We consider a simplified object-oriented language, i.e., a subset of Java, in which a program consists of a set of classes, where each class consists of static/instance fields and methods. Table 2 gives six kinds of statements, which are labeled by their line numbers, in the language operated on by kOBJ. Note that "x = **new** T(...)" in Java is modeled as "x = **new** T; x.⟨`init`⟩(...)", where ⟨`init`⟩() is the corresponding constructor invoked. Section 5 discusses how to handle other complex language features such as reflection and native code.

As kOBJ is context-sensitive but flow-insensitive, the control flow statements in a program are irrelevant. As is standard with several recent implementations of kOBJ [31, 39, 41, 12],

Kind	Statement	Description
new	$l : v = \textbf{new}\ T$	v is a local variable and T is a class type
assign	$l : v = v'$	v and v' are local variables
assignglobal	$l : v = v'$	v or v' is a global variable
load	$l : v = v'.f$	v and v' are local variables and f is a field name
store	$l : v.f = v'$	v and v' are local variables and f is a field name
call	$l : b = a_0.m(a_1, ..., a_r)$	b and a_i are local variables and m is an instance method

Table 2 Six types of statements analyzed by kOBJ.

static fields are analyzed context-insensitively as global variables, but static methods can be analyzed context-sensitively as instance methods as follows. For a static method $m()$ defined in class C, a call to $m()$ can be interpreted as $this.m()$ by proceeding as if $m()$ were an instance method defined in `java.lang.Object` and inherited by C. Given $this.m()$, $m()$ can then be analyzed context-sensitively under the receiver object pointed to by $this$, which is the receiver object of m's closest (instance) caller method, if any, on the call stack.

Finally, every method is assumed to have one single return statement of the form "**return** r", where r is a local variable (referred to as its return variable). Note that a return statement in a method is not listed explicitly in Table 2, as it will be handled implicitly at a call statement where the method is invoked (as shown in Figure 4).

3.2 Selective Object-Sensitive Pointer Analysis

Given a program, let $\mathbb{M}$, $\mathbb{F}$, $\mathbb{H}$, $\mathbb{V}$, $\mathbb{G}$ and $\mathbb{L}$ be its sets of methods, fields, allocation sites, local variables, global variables, and statements (identified by their labels, e.g., line numbers), respectively.

Let $\mathbb{C} = \mathbb{H}^*$ be the universe of contexts. Given a context $ctx = [o_1, ..., o_n] \in \mathbb{C}$ and a context element o, we write $o +\!\!+\ ctx$ for $[o, o_1, ..., o_n]$ and $\lceil ctx \rceil_k$ for $[o_1, ..., o_k]$.

The rules used for performing kOBJ will make use of the following functions:

- $\mathsf{methodOf} : \mathbb{L} \mapsto \mathbb{M}$
- $\mathsf{methodCtx} : \mathbb{M} \mapsto \wp(\mathbb{C})$
- $\mathsf{dispatch} : \mathbb{M} \times \mathbb{H} \mapsto \mathbb{M}$
- $\mathsf{len} : \mathbb{V} \cup \mathbb{G} \cup \mathbb{H} \mapsto \mathbb{N}$
- $\mathsf{pts} : (\mathbb{V} \cup \mathbb{H} \times \mathbb{F}) \times \mathbb{C} \mapsto \wp(\mathbb{H} \times \mathbb{C})$

where methodOf gives the containing method of a statement, methodCtx keeps track of the (method) contexts used for analyzing a method, dispatch resolves a virtual call to its target method, len defines the length of contexts used for analyzing a variable/object, and pts records the points-to information found for a variable or an object's field.

Figure 4 gives five rules used by kOBJ for analyzing six kinds of statements in Table 2 with two kinds of assignments processed together in one rule. In [New], v points to the object o_l uniquely identified by its allocation site l. Note that $\lceil ctx \rceil_{\mathsf{len}(o_l)}$ is the heap context of o_l (Section 2.1). In [Assign/AssignGlobal], two kinds of assignments, where v and v' are either local or global variables, are handled as copies. In [Store] and [Load], field accesses are analyzed in the standard manner. In [CALL], a call to an instance method $b = a_0.m(a_1, ..., a_r)$ is analyzed. 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, $ctx' \in \mathsf{methodCtx}(m')$ reveals how the method contexts of a method are introduced. Initially, $\mathsf{methodCtx}(\text{“}\texttt{main}\text{”}) = \{[\,]\}$.

$$\frac{l : v = \text{new } T \quad M = \mathsf{methodOf}(l) \quad ctx \in \mathsf{methodCtx}(M)}{(o_l, \lceil ctx \rceil_{\mathsf{len}(o_l)}) \in \mathsf{pts}(v, \lceil ctx \rceil_{\mathsf{len}(v)})} \quad \text{[New]}$$

$$\frac{l : v = v' \quad M = \mathsf{methodOf}(l) \quad ctx \in \mathsf{methodCtx}(M)}{\mathsf{pts}(v', \lceil ctx \rceil_{\mathsf{len}(v')}) \subseteq \mathsf{pts}(v, \lceil ctx \rceil_{\mathsf{len}(v)})} \quad \text{[Assign/AssignGlobal]}$$

$$\frac{l : v.f = v' \quad M = \mathsf{methodOf}(l) \quad ctx \in \mathsf{methodCtx}(M) \quad (o, hctx) \in \mathsf{pts}(v, \lceil ctx \rceil_{\mathsf{len}(v)})}{\mathsf{pts}(v', \lceil ctx \rceil_{\mathsf{len}(v')}) \subseteq \mathsf{pts}(o.f, hctx)} \quad \text{[Store]}$$

$$\frac{l : v = v'.f \quad M = \mathsf{methodOf}(l) \quad ctx \in \mathsf{methodCtx}(M) \quad (o, hctx) \in \mathsf{pts}(v', \lceil ctx \rceil_{\mathsf{len}(v')})}{\mathsf{pts}(o.f, hctx) \subseteq \mathsf{pts}(v, \lceil ctx \rceil_{\mathsf{len}(v)})} \quad \text{[Load]}$$

$$\frac{\begin{array}{c} l : b = a_0.m(a_1, ..., a_r) \quad M = \mathsf{methodOf}(l) \quad ctx \in \mathsf{methodCtx}(M) \\ (o, hctx) \in \mathsf{pts}(a_0, \lceil ctx \rceil_{\mathsf{len}(a_0)}) \quad m' = \mathsf{dispatch}(m, o) \quad ctx' = o +\!\!+ \; hctx \end{array}}{\begin{array}{c} ctx' \in \mathsf{methodCtx}(m') \quad (o, hctx) \in \mathsf{pts}(this^{m'}, \lceil ctx' \rceil_{\mathsf{len}(this^{m'})}) \\ \forall i \in [1, r] : \mathsf{pts}(a_i, \lceil ctx \rceil_{\mathsf{len}(a_i)}) \subseteq \mathsf{pts}(p_i^{m'}, \lceil ctx' \rceil_{\mathsf{len}(p_i^{m'})}) \quad \mathsf{pts}(ret^{m'}, \lceil ctx' \rceil_{\mathsf{len}(ret^{m'})}) \subseteq \mathsf{pts}(b, \lceil ctx \rceil_{\mathsf{len}(b)}) \end{array}} \quad \text{[Call]}$$

Figure 4 Rules for kOBJ formalized to support selective context-sensitivity.

kOBJ represents a k-object-sensitive pointer analysis with a $(k-1)$-context-sensitive heap (by handling global variables context-insensitively as is standard) [31, 39, 41, 12]. Thus, kOBJ selects the context lengths for different entities e in $\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}$ differently as follows:

$$\mathsf{len}_{k\text{OBJ}}(e) = \begin{cases} 0 & e \in \mathbb{G} \\ k & e \in \mathbb{V} \\ k-1 & e \in \mathbb{H} \end{cases} \tag{1}$$

As a pre-analysis, TURNER will select a subset $CI_{\text{TURNER}} \subseteq \mathbb{V} \cup \mathbb{H}$ so that kOBJ will analyze CI_{TURNER} context-insensitively but $(\mathbb{V} \cup \mathbb{H}) \setminus CI_{\text{TURNER}}$ context-sensitively as follows:

$$\mathsf{len}_{\text{TURNER}}(e) = \begin{cases} 0 & e \in CI_{\text{TURNER}} \\ \mathsf{len}_{k\text{OBJ}}(e) & e \in (\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}) \setminus CI_{\text{TURNER}} \end{cases} \tag{2}$$

As discussed earlier, EAGLE [22] will also enable kOBJ to analyze only a subset of variables/objects in a method context-sensitively but ZIPPER [19] will require a method (i.e., all its variables/objects) to be analyzed either fully context-sensitively or fully context-insensitively.

4 Turner: Our Approach

We describe the two stages of TURNER, object containment (Section 4.1) and reachability (Section 4.2), by focusing predominantly on formalizing our object reachability analysis.

4.1 Object Containment

In this first stage on object containment analysis, we identify some precision-uncritical objects in a program based on the points-to information pre-computed by Andersen's analysis [1] according to Observation 3. For an object o, we write ret_o to denote the return variable in the method where o is allocated. For two objects o_1 and o_2, we write $o_1 \xrightarrow{class-type(f)} o_2$ if $o_1.f = o_2$ for some field f whose declared type is either C or $C[\,]$, where C is some class type. As a result, the set of precision-uncritical objects in a program can be found by:

$$\mathsf{CI}_{\text{TURNER}}^{\text{OBS}} = \mathsf{TopCon} \cup \mathsf{BotCon} \tag{3}$$

where the sets of top and bottom containers in the program are identified as follows:

$$\begin{aligned} \mathsf{TopCon} &= \left\{ o \,\middle|\, \left(\nexists\ (o', f) \in \mathbb{H} \times \mathbb{F} : o' \xrightarrow{class-type(f)} o \right) \land ret_o \text{ does not point to } o \right\} \\ \mathsf{BotCon} &= \left\{ o \,\middle|\, \nexists\ (o', f) \in \mathbb{H} \times \mathbb{F} : o \xrightarrow{class-type(f)} o' \right\} \end{aligned} \tag{4}$$

4.2 Object Reachability

In this second stage on object reachability analysis, we make use of a DFA to determine intra-procedurally whether a variable/object requires context-sensitivity or not. Let CI_{TURNER} be the set of context-insensitive variables/objects that are finally selected by TURNER to support selective context-sensitivity required in (2). By design, $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} \subseteq CI_{\text{TURNER}}$, i.e., the precision-uncritical objects selected earlier will always be analyzed context-insensitively. The remaining objects and all the variables in the program will be further classified as either context-sensitive or context-insensitive according to the DFA, by leveraging $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$.

We first review a standard formulation for performing pointer analysis intra-procedurally based on CFL (Context-Free Language) reachability (Section 4.2.1). We then evolve it incrementally into a DFA-based intra-procedural reachability analysis (Section 4.2.2).

4.2.1 Standard CFL-Reachability-based Pointer Analysis

A parameterless method that contains no calls inside can be represented by a directed graph G, known as PAG (Pointer Assignment Graph), with its nodes drawn from $\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}$ and its five types of edges added according to the rules given in Figure 5 [37, 28]. Loads and stores to the elements of an array are modeled by collapsing all the elements into a special field `arr` of the array. For each PAG edge $x \xrightarrow{\ell} y$ with its label ℓ, its inverse edge is denoted as $y \xrightarrow{\overline{\ell}} x$.

$$\frac{l : v = \textbf{new}\ T}{o_l \xrightarrow{\mathsf{new}} v \quad v \xrightarrow{\overline{\mathsf{new}}} o_l}\ \text{[P-NEW]} \qquad \frac{v\ =\ v'.f}{v' \xrightarrow{\mathsf{load[f]}} v \quad v \xrightarrow{\overline{\mathsf{load[f]}}} v'}\ \text{[P-LOAD]} \qquad \frac{v.f\ =\ v'}{v' \xrightarrow{\mathsf{store[f]}} v \quad v \xrightarrow{\overline{\mathsf{store[f]}}} v'}\ \text{[P-STORE]}$$

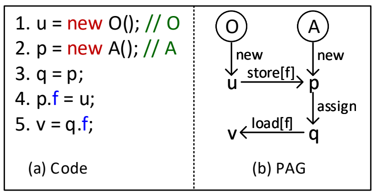
$$\frac{v\ =\ v'}{v' \xrightarrow{\mathsf{assign}} v \quad v \xrightarrow{\overline{\mathsf{assign}}} v'}\ \text{[P-ASSIGN]} \qquad \frac{v\ =\ v'}{v' \xrightarrow{\mathsf{assignglobal}} v \quad v \xrightarrow{\overline{\mathsf{assignglobal}}} v'}\ \text{[P-ASSIGNGLOBAL]}$$

Figure 5 Rules for creating the PAG edges for a method containing no calls inside.

Let L be a CFL over Σ formed by the edge labels in G. Each path p in G has a string $L(p)$ in Σ^* formed by concatenating in order the labels of edges in p. A node v in G is *L-reachable* from a node u in G if there exists a path p from u to v, known as *L-path* and denoted by $L(u, v)$, such that $L(p) \in L$. For a node n in G, we write $L(u, v)^n$ if n appears on $L(u, v)$. For a path p in G such that its label is $L(p) = \ell_1, \cdots, \ell_r$ in L, the inverse of p, i.e., $\overline{p}$ has the label $L(\overline{p}) = \overline{\ell_r}, \cdots, \overline{\ell_1}$.

We start with a standard grammar that defines the following language L_0 [37, 28]:

$$L_0 : \begin{cases} \mathit{flowsto} \longrightarrow \mathsf{new}\ \mathit{flows}^* \\ \mathit{flows} \longrightarrow \mathsf{assign} \mid \mathsf{assignglobal} \mid \mathsf{store[f]}\ \mathit{alias}\ \mathsf{load[f]} \\ \overline{\mathit{flowsto}} \longrightarrow \overline{\mathit{flows}}^*\ \overline{\mathsf{new}} \\ \overline{\mathit{flows}} \longrightarrow \overline{\mathsf{assign}} \mid \overline{\mathsf{assignglobal}} \mid \overline{\mathsf{load[f]}}\ \mathit{alias}\ \overline{\mathsf{store[f]}} \\ \mathit{alias} \longrightarrow \overline{\mathit{flowsto}}\ \mathit{flowsto} \end{cases} \tag{5}$$



```
u = new O(); // O
p = new A(); // A
q = p;
p.f = u;
v = q.f;
```

Figure 6 The PAG for a code snippet.

If o *flowsto* v, then v is L_0-reachable from o, i.e., $L_0(o, v)$. To handle aliases, $\overline{\mathit{flowsto}}$ is introduced as the inverse of the *flowsto* relation. A *flowsto* path p can be inverted to obtain its corresponding $\overline{\mathit{flowsto}}$ path $\overline{p}$ using its inverse edges, and vice versa. Thus, o *flowsto* x iff x $\overline{\mathit{flowsto}}$ o. This means that $\overline{\mathit{flowsto}}$ actually represents the standard points-to relation. As a result, x *alias* y iff x $\overline{\mathit{flowsto}}$ o *flowsto* y for some object o, so that field accesses are handled precisely by solving a *balanced parentheses* problem. For the code snippet (consisting of local variables only), together with its PAG, depicted in Figure 6, we know that $L_0(O, v)$, i.e., `O` *flowsto* `v`, implying that `v` points to `O`, which holds due to the following *flowsto* path:

$$\texttt{O} \xrightarrow{\textsf{new}} \texttt{u} \xrightarrow{\textsf{store[f]}} \texttt{p} \xrightarrow{\overline{\textsf{new}}} \texttt{A} \xrightarrow{\textsf{new}} \texttt{p} \xrightarrow{\textsf{assign}} \texttt{q} \xrightarrow{\textsf{load[f]}} \texttt{v} \tag{6}$$

By inverting all the edges in this *flowsto* path, a $\overline{\mathit{flowsto}}$ path showing `v` $\overline{\mathit{flowsto}}$ `O` is obtained.

4.2.2 Turner's Context-Sensitivity-Deciding Reachability Analysis

We will now over-approximate L_0 incrementally to obtain a regular grammar, i.e., a DFA to decide intra-procedurally whether a variable/object requires context-sensitivity or not.

4.2.2.1 Ignoring Context-Insensitive Value Flows

Instead of computing points-to information in a program directly, Turner is designed to analyze the context-sensitive value flows across the parameters or return variables of its methods (Fact 2). Thus, we will ignore the assignglobal statements and the precision-uncritical objects in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$, as all the value-flows passing through them are context-insensitive.

$$\frac{l : v = \textbf{new } T \quad o_l \notin \mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}}{o_l \xrightarrow{\textsf{cs-likely}} o_l} \quad [\text{P-Object}]$$

Figure 7 Rule for treating all the objects in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$ as context-insensitive.

To handle the objects in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$ context-insensitively as global variables, as shown in Figure 7, we have added a self-loop edge label, named cs-likely, for each object that is not in $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{TURNER}}$ to indicate that it is currently treated as being potentially context-sensitive but will be classified later as being either context-sensitive or context-insensitive by our reachability analysis. By deleting the two terminals assignglobal and $\overline{\textsf{assignglobal}}$ from and adding one new terminal cs-likely to the grammar for defining L_0, we obtain:

$$L_1 : \begin{cases} \mathit{flowsto} \longrightarrow \textsf{new}\ \mathit{flows}^* \\ \mathit{flows} \longrightarrow \textsf{assign} \mid \textsf{store[f]}\ \mathit{alias}\ \textsf{load[f]} \\ \overline{\mathit{flowsto}} \longrightarrow \overline{\mathit{flows}}^*\ \overline{\textsf{new}} \\ \overline{\mathit{flows}} \longrightarrow \overline{\textsf{assign}} \mid \overline{\textsf{load[f]}}\ \mathit{alias}\ \overline{\textsf{store[f]}} \\ \mathit{alias} \longrightarrow \overline{\mathit{flowsto}}\ \textsf{cs-likely}\ \mathit{flowsto} \end{cases} \tag{7}$$

We will discuss how to handle method parameters and method calls shortly below.

Let us consider Figure 6 again by making two independent changes to the code snippet:

- If `q` is a global variable, then $\mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q}$ will become $\mathtt{p} \xrightarrow{\mathsf{assignglobal}} \mathtt{q}$. As a result, $L_1(\mathtt{O}, \mathtt{v})$ can no longer be established as in (6) earlier (due to the absence of assignglobal in L_1).
- if `A` is a cs-likely object, then $L_1(\mathtt{O}, \mathtt{v})$ can also be established as before, since we have:

$$\mathtt{O} \xrightarrow{\mathsf{new}} \mathtt{u} \xrightarrow{\mathsf{store[f]}} \mathtt{p} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q} \xrightarrow{\mathsf{load[f]}} \mathtt{v} \tag{8}$$

Otherwise, $L_1(\mathtt{O}, \mathtt{v})$ will no longer be possible due to the absence of $\mathtt{A} \xrightarrow{\mathsf{cs\text{-}likely}} \mathtt{A}$.

To simplify matters, returning values from a method can be treated identically as passing parameters for the method. In object-sensitive pointer analysis [31, 39, 41, 12, 22, 21], a method M is analyzed context-sensitively under different receiver objects. Thus, its return statement "**return** r" can be modeled as "*this.ret* = r", where *ret* is a fresh local variable (interpreted now as the *return variable* of M) and the return values in "*this.ret*" can be retrieved by its callers via its receiver objects. Given this simple transformation, the four value-flow patterns given in Figure 1 can be unified as one "*param-param*" pattern.

▶ **Lemma 5.** *A variable/object n in a method M requires context-sensitivity only if n lies on a sequence of statements, $s_1, ..., s_r$, such that (1) s_i and s_{i+1} form a def-use chain involving only local variables and cs-likely objects, (2) s_1 represents a use of either a cs-likely object or a parameter of M, and (3) s_r represents a definition of $P.f$, where P is a parameter of M (including this) and f is a field of the objects pointed by P (including M's return variable (ret)).*

Proof. Follows directly from Fact 2 and the definition of cs-likely objects. ◀

In this case, n should be context-sensitive, since the modification effects of different definitions of n on $P.f$ under different calling contexts of M must be separated context-sensitively.

4.2.2.2 Approximating the Value Flows Spanning across Method Calls

We now consider how to handle a method call made in a method being analyzed. TURNER will over-approximate the context-sensitive value flows spanning across a call site without analyzing its invoked methods. With L_1, we can only reason about CFL reachability starting from an object. With L_2 given below, we can also start from a variable (Lemma 5):

$$L_2 : \begin{cases} \mathit{flows} \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{store[f]}\ \mathit{alias}\ \mathsf{load[f]})^* \\ \overline{\mathit{flows}} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load[f]}}\ \mathit{alias}\ \overline{\mathsf{store[f]}})^* \\ \mathit{alias} \longrightarrow \overline{\mathit{flows}}\ \mathsf{cs\text{-}likely}\ \mathit{flows} \end{cases} \tag{9}$$

▶ **Lemma 6.** *Let G be the PAG built by the rules in Figures 5 and 7. $L_2 \supseteq L_1$.*

Proof. Follows simply from examining the structural differences in their productions. ◀

In both languages, the aliases between two variables are established in exactly the same way.

Next, we over-approximate L_2 to obtain L_3 by abstracting the field accesses with 1-limited access paths and handling aliases more conservatively (as explained shortly below):

$$L_3 : \begin{cases} \mathit{flows} \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{load} \mid \mathsf{store}\ \mathit{alias})^* \\ \overline{\mathit{flows}} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load}} \mid \mathit{alias}\ \overline{\mathsf{store}})^* \\ \mathit{alias} \longrightarrow \overline{\mathit{flows}}\ \mathsf{cs\text{-}likely}\ \mathit{flows} \end{cases} \tag{10}$$

Thus, the fields in loads and stores are ignored, and loads and assignments become indistinguishable, but stores are treated differently (i.e., unsymmetrically as loads) in order to keep track of aliases as desired. Note that L_3 is is still a CFL, since (1) a store is required to match a $\overline{\textsf{new}}$, $\overline{\textsf{assign}}$ or $\overline{\textsf{load}}$, and (2) a $\overline{\textsf{store}}$ is required to match a new, assign or load. This balanced-parentheses property is somehow hidden in the *alias*-production.

For the code given in Figure 6, $L_3(\texttt{O}, \texttt{v})$ will still hold even if, say, `v = q.f` is replaced by `v = q.g` due to the existence of the following *flowsto* path:

$$\texttt{O} \xrightarrow{\textsf{new}} \texttt{u} \xrightarrow{\textsf{store}} \texttt{p} \xrightarrow{\overline{\textsf{new}}} \texttt{A} \xrightarrow{\textsf{cs-likely}} \texttt{A} \xrightarrow{\textsf{new}} \texttt{p} \xrightarrow{\textsf{assign}} \texttt{q} \xrightarrow{\textsf{load}} \texttt{v} \tag{11}$$

▶ **Lemma 7.** *Let G be the PAG built by the rules in Figures 5 and 7. $L_3 \supseteq L_2$.*

Proof. In L_3, the first two productions can be expressed equivalently as $\mathit{flows} \longrightarrow (\textsf{new} \mid \textsf{assign} \mid \textsf{load} \mid \textsf{store}\ \mathit{alias}\ \textsf{load}?)^*$ and $\overline{\mathit{flows}} \longrightarrow (\overline{\textsf{new}} \mid \overline{\textsf{assign}} \mid \overline{\textsf{load}} \mid \overline{\textsf{load}}?\ \mathit{alias}\ \overline{\textsf{store}})^*$. Here, (s)? indicates that s is optional, where '(' and ')' can be omitted if s represents one symbol. We can conclude that $L_3 \supseteq L_2$ by noting that the field access paths in L_3 are 1-limited. ◀

In L_3, a store can now also be matched with a $\overline{\textsf{store}}$ when looking for aliases:

$$\mathit{flows} \Longrightarrow^+ \dots\ \textsf{store}\ \overline{\mathit{flows}}\ \textsf{cs-likely}\ \mathit{flows}\ \overline{\textsf{store}}\ \dots \tag{12}$$

For the code given in Figure 6, $L_3(\texttt{O}, \texttt{v})$ will thus still hold if we (1) replace `v = q.f` by `q.g = v` and (2) add `v = new V()`, where the allocated object, `V`, is assumed to be cs-likely:

$$\texttt{O} \xrightarrow{\textsf{new}} \texttt{u} \xrightarrow{\textsf{store}} \texttt{p} \xrightarrow{\overline{\textsf{new}}} \texttt{A} \xrightarrow{\textsf{cs-likely}} \texttt{A} \xrightarrow{\textsf{new}} \texttt{p} \xrightarrow{\textsf{assign}} \texttt{q} \xrightarrow{\overline{\textsf{store}}} \texttt{v} \xrightarrow{\overline{\textsf{new}}} \texttt{V} \xrightarrow{\textsf{cs-likely}} \texttt{V} \xrightarrow{\textsf{new}} \texttt{v} \tag{13}$$

We discuss below how to exploit this property to avoid analyzing the methods invoked at a call site while still keeping track of all context-sensitive value flows spanning the call site.

$$\frac{b\ =\ a_0.m(a_1, \dots, a_r)}{\forall\ i : a_i \xrightarrow{\textsf{store}[p_i^{m'}]} a_0 \quad \forall\ i : a_0 \xrightarrow{\overline{\textsf{store}[p_i^{m'}]}} a_i \quad a_0 \xrightarrow{\textsf{load}[ret^{m'}]} b \quad b \xrightarrow{\overline{\textsf{load}[ret^{m'}]}} a_0} \ \text{[P-CALL]}$$

Figure 8 Rule for analyzing a method call.

Consider how kOBJ analyzes a method call $b = a_0.m(a_1, \dots, a_r)$, with a target method m' resolved when a_0 points to a receiver object O. Let its $r+1$ parameters be $p_0^{m'}, \dots, p_r^{m'}$, where $p_0^{m'}$ represents $\mathit{this}^{m'}$. Let its return variable $\mathit{ret}^{m'}$ be introduced as described in Section 4.2.2.1. Object-sensitively, $p_0^{m'}, \dots, p_r^{m'}$ and $\mathit{ret}^{m'}$ are handled as if they were special fields of O [22, 21]: $\forall\ i : a_0.p_i^{m'} = a_i$ for passing parameters and $b = a_0.\mathit{ret}^{m'}$ (for retrieving return values). As a result, Figure 8 gives a rule, [P-CALL], for adding the PAG edges required for a method call according to [P-LOAD] and [P-STORE]. When m' is analyzed by kOBJ, where its $\mathit{this}^{m'}$ variable points to O, its parameters will be initialized as $\forall\ i : p_i^{m'} = \mathit{this}^{m'}.p_i^{m'}$ and its return values will be made available in $\mathit{this}^{m'}.\mathit{ret}^{m'}$.

Given how $b = a_0.m(a_1, \dots, a_r)$ is modeled above, we can determine whether or not a context-sensitive value flow that enters one of its invoked methods via a parameter can also exit it via another parameter without actually analyzing the invoked method itself, by enforcing $L_3(a_i, a_j)$ conservatively, i.e., ensuring that whatever flows into a_i flows also into a_j, if necessary. As will be clear in Section 4.2.2.3 below, $b = a_0.m(a_1, \dots, a_r)$ needs to be approximated this way if a_0 may point to at least one cs-likely object and can be ignored otherwise.

▶ **Lemma 8.** *Let G be the PAG built by the rules in Figures 5, 7 and 8 for a method M (where how its parameters are modeled is irrelevant here). When analyzing a call $b = a_0.m(a_1, \ldots, a_r)$ contained in M, $L_3(a_i, a_j)$ is established iff a_0 points to at least one* cs-likely *object.*

Proof. Let O be an object pointed by a_0. By [P-CALL], passing a_i and a_j to a target method m' at the call site is modeled by two stores $a_0.p_i^{m'} = a_i$ and $a_0.p_i^{m'} = a_j$. Thus, we have:

$$flows \Longrightarrow^{+} \ldots\ a_i \xrightarrow{\mathsf{store}} a_0\ \overline{flows}\ O \cdots O\ flows\ a_0 \xrightarrow{\overline{\mathsf{store}}} a_j\ \ldots \tag{14}$$

As a result, $L_3(a_i, a_j)$ is established (as far as this particular call site is concerned, regardless of its truthhood established elsewhere) iff O is a cs-likely object, in which case the "$\cdots$" that sits between the two occurrences of O can be replaced by $\xrightarrow{\text{cs-likely}}$. ◀

4.2.2.3 Approximating the Incoming Value Flows from Parameters

We discuss now how to handle the parameters of a method when it is analyzed. It is not computationally feasible to formulate our pre-analysis for a method in terms of L_3 directly (even after its parameters are modeled in a certain way). As L_3 is a CFL (with balanced parentheses), the worst-time complexity for finding the points-to set of a variable is $O(N^3\Gamma_{L_3}^3)$, where N is the number of nodes in the PAG and Γ_{L_3} is the size of L_3 [26, 15].

We now over-approximate L_3 by turning it into a regular language L_4 defined by:

$$L_4 : \left\{ \begin{array}{l} flows \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{load})^{*}((\mathsf{store} \mid \overline{\mathsf{store}})\ \overline{flows})? \\ \overline{flows} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load}})^{*}(\text{cs-likely}\ flows)? \end{array} \right. \tag{15}$$

▶ **Lemma 9.** *Let G be the PAG built by the rules in Figures 5, 7 and 8. $L_4 \supseteq L_3$.*

Proof. L_4 is regularized from L_3 by no longer distinguishing store and $\overline{\text{store}}$. ◀

Thus, we are now even more conservative in abstracting aliases in L_4 than in L_3. If we replace `p.f = u` with `u.f = p` in Figure 6, $L_3(\mathtt{O}, \mathtt{v})$ will not hold but $L_4(\mathtt{O}, \mathtt{v})$ will, since

$$\mathtt{O} \xrightarrow{\mathsf{new}} \mathtt{u} \xrightarrow{\overline{\mathsf{store}}} \mathtt{p} \xrightarrow{\overline{\mathsf{new}}} \mathtt{A} \xrightarrow{\text{cs-likely}} \mathtt{A} \xrightarrow{\mathsf{new}} \mathtt{p} \xrightarrow{\mathsf{assign}} \mathtt{q} \xrightarrow{\mathsf{load}} \mathtt{v} \tag{16}$$

$$\frac{p \text{ is a parameter}}{p \xrightarrow{\mathsf{param}} p \quad p \xrightarrow{\overline{\mathsf{param}}} p}\ \text{[P-PARAM]}$$

Figure 9 Rule for adding the PAG edges for parameters.

We are now ready to describe our final regular language L_5 used to decide if a variable/object in a method should be context-sensitive or not. By exploiting the fact that store and $\overline{\text{store}}$ are treated identically in L_4, we have obtained L_5, requiring the two self-loop edges to be added for each parameter of a method according to a rule, [P-PARAM], given in Figure 9:

$$L_5 : \left\{ \begin{array}{l} s \longrightarrow \mathsf{param}\ flows \\ flows \longrightarrow (\mathsf{new} \mid \mathsf{assign} \mid \mathsf{load})^{*}((\mathsf{store} \mid \overline{\mathsf{store}})\ \overline{flows})? \\ \overline{flows} \longrightarrow (\overline{\mathsf{new}} \mid \overline{\mathsf{assign}} \mid \overline{\mathsf{load}})^{*}(\text{cs-likely}\ flows)? \\ \overline{flows} \longrightarrow \overline{\mathsf{param}}\ e \\ e \longrightarrow \epsilon \end{array} \right. \tag{17}$$

We can now analyze a method without knowing what its parameters may point to, by treating it effectively as a parameterless method, so that all the results developed so far are applicable.

▶ **Lemma 10.** *Let G be the PAG built for a method by the rules in Figures 5 and 7–9. Let P_1 and P_2 be its two (not necessarily different) parameters. Then $L_4(P_1, P_2) \iff L_5(P_1, P_2)$.*

Proof. Follows straightforwardly by noting the minor differences in their productions. ◀

As discussed in Section 4.2.1, if L is a CFL, $L(u, v)^n$ holds if $L(u, v)$ holds due to an L-path that contains a node n. Thus, CI_{TURNER} that appears in (2) can now be defined as:

$$CI_{\text{TURNER}} = \{n \mid M \in \mathbb{M}, n \text{ is a node in } G_M, \nexists P_1, P_2 \in \mathit{param}(M) : L_5^{G_M}(P_1, P_2)^n\} \quad (18)$$

where $\mathit{param}(M)$ is the set of parameters of a method M and L_5 is superscripted with the PAG, G_M, built for M. By construction, $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} \subseteq CI_{\text{TURNER}}$ holds due to the absence of a self-loop edge, labeled cs-likely, around each object in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$. In addition, $\mathbb{G} \subseteq CI_{\text{TURNER}}$. However, all the global variables will be context-insensitive according to (1) regardless.

Let us apply TURNER to the four examples in Figure 1 to see how it has successfully selected `x` to be context-sensitive (where "return `x`" in each example has been replaced by "`this.ret = x`" and the object `A` created in Figure 1(d) is assumed to be a cs-likely object):

- **Figures 1(a) and 1(b):** $L_5(\texttt{p}, \texttt{this})^{\texttt{x}}$: $\texttt{p} \xrightarrow{\text{assign}} \texttt{x} \xrightarrow{\text{store}} \texttt{this}$.
- **Figure 1(c).** $L_5(\texttt{p}, \texttt{q})^{\texttt{x}}$: $\texttt{p} \xrightarrow{\text{assign}} \texttt{x} \xrightarrow{\overline{\text{store}}} \texttt{q}$.
- **Figure 1(d):** $L_5(\texttt{this}, \texttt{this})^{\texttt{x}}$: $\texttt{this} \xrightarrow{\overline{\text{store}}} \texttt{x} \xrightarrow{\overline{\text{new}}} \texttt{A} \xrightarrow{\text{cs-likely}} \texttt{A} \xrightarrow{\text{new}} \texttt{x} \xrightarrow{\text{store}} \texttt{this}$.

Finally, we show that TURNER is precision-preserving if Observation 3 is precision-preserving. The basic idea is to show that if a variable/object is context-sensitive according to Lemma 5, i.e., Fact 2 (Figure 1), then it must reside on an L_5-path.

▶ **Theorem 11.** *Suppose Observation 3 is precision-preserving. Let G be the PAG built for a method M (Figures 5 and 7–9). If a variable/object n in M is context-sensitive by Lemma 5, then $L_5(P_1, P_2)^n$, where P_1 and P_2 are two (not necessarily different) parameters of M.*

Proof. Our proof proceeds in the following three steps:

1. We assume that M is analyzed equivalently under one cs-likely receiver object, O_M. Let M' be obtained from M by augmenting it with (1) "this^M = **new** T // O_M" and (2) "$P = \mathit{this}^M.P$" for every parameter P of M. Let G' be the resulting PAG augmented from G. For every parameter P of M, we now have $P \xrightarrow{\overline{\text{assign}}} \mathit{this}^M \xrightarrow{\overline{\text{new}}} O_M \xrightarrow{\text{cs-likely}} O_M \xrightarrow{\text{new}} \mathit{this}^M \xrightarrow{\text{assign}} P$. Thus, $L_5(P_1, P_2)^n$ holds over G, where P_1 and P_2 are two parameters of M iff $L_5(P', P')^n$ holds over G', where P' is a parameter of M. In L_5, every variable will now be guaranteed to point to at least one object, which can be O_M.
2. We show now that all the context-sensitive value flows that enter M under its different calling contexts are tracked in L_5 if they pass through a method call $b = a_0.m_0(a_1, ..., a_r)$ (via $a_0, ..., a_r$). Thus, it suffices to consider each call site in M in isolation. Note that the loads and stores in a program can always be modeled as getters and setters.
   By Lemmas 9 and 10, Lemma 8 applies also to L_5: $L_5(a_i, a_j)$ is established in analyzing $b = a_0.m_0(a_1, ..., a_r)$ iff a_0 points to at least one cs-likely object. Thus, we only need to argue that if a_0 points to only context-insensitive objects, recorded in F_{a_0}, then each invoked method at this call site can be ignored in this sense. In this case (where $O_M \notin F_{a_0}$ as O_M is context-sensitive by construction), if some pointed-to objects of a_0 are missing in F_{a_0} (as our pre-analysis is intra-procedural), then there must exist a

call chain, $a_0 = x_1.m_1(...), x_1 = x_2.m_2(...), ..., x_{t-1} = x_t.m_t(...)$ (modeled effectively as $a_0 = x_1 = ... = x_t$ in L_5), where all the pointed-to objects of x_t in the program are found intra-procedurally (under the assumption that all the receiver objects of M are abstracted by one single context-sensitive object, O_M, as explained in Step 1).
Since Observation 3 is assumed to be precision-preserving, the value flows that enter M under its different calling contexts (i.e., receiver objects) need not be tracked, i.e., separated context-sensitively at each call site $m_i()$. To prove this claim inductively, let us write $x_{-1} = x_0.m_0(...)$ to represent $b = a_0.m_0(...)$. Now, let R_{m_i} be the set of objects returned by $m_i()$ but missed by L_5, as $m_i()$ is not analyzed. Our claim is true for $x_{t-1} = x_t.m_t(...)$, since all the objects pointed to by x_t in the program are context-insensitive. This also implies that the objects in R_{m_t} are all conflated under different calling contexts of M. Suppose that our claim holds for $m_i()$, in which case, the objects in R_{m_i} are all conflated. Let us consider $x_{i-2} = x_{i-1}.m_{i-1}(...)$. As x_{i-1} can only point to either some context-insensitive objects in F_{a_0} found intra-procedurally by L_5 or the conflated objects in R_{m_i}, our claim must also be true for $m_{i-1}()$.

3. If a variable n is context-sensitive by Lemma 5, there must exist a cs-likely O due to Step 1 such that $L_1(O,P)^n : O\ \mathit{flows}\ n' \xrightarrow{\text{store}} P$, which contains n, where n' is a variable (which may be n) and P is a parameter of M. By applying Lemmas 6 – 10 and the result established in Step 2, we must have $L_5(O,P)^n : O\ \mathit{flows}\ n' \xrightarrow{\text{store}} P$ (passing through n). As a result, $L_5(P,P)^n : P \xrightarrow{\overline{\text{store}}} n'\ \overline{\mathit{flows}}\ O \xrightarrow{\text{cs-likely}} O\ \mathit{flows}\ n' \xrightarrow{\text{store}} P$ holds. If an object n is context-sensitive by Lemma 5, $L_5(P,P)^n$ can be established similarly. ◀

4.2.2.4 Computing CI_{Turner} with a DFA

We give an efficient algorithm for computing CI_{TURNER} with a DFA (Figure 10) obtained equivalently from the regular grammar for L_5. Our algorithm proceeds in linear time of the number of nodes in the PAG by exploiting an antisymmetric property in our DFA.

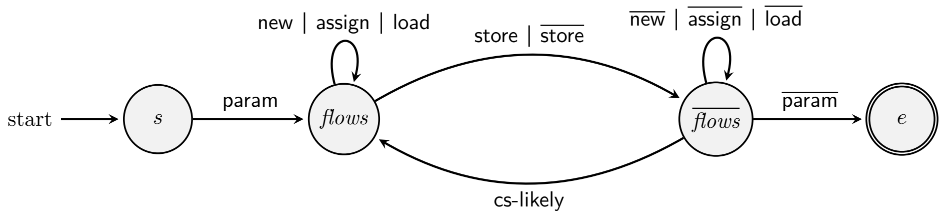

new | assign | load
store | $\overline{\text{store}}$
$\overline{\text{new}}$ | $\overline{\text{assign}}$ | $\overline{\text{load}}$
start
s
param
flows
$\overline{\mathit{flows}}$
$\overline{\text{param}}$
e
cs-likely

Figure 10 The DFA as an equivalent representation of the grammar for defining L_5.

The DFA is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, s, e)$, where $Q = \{s, \mathit{flows}, \overline{\mathit{flows}}, e\}$ is the set of states, $\Sigma = \{\text{param}, \overline{\text{param}}, \text{new}, \overline{\text{new}}, \text{assign}, \overline{\text{assign}}, \text{load}, \overline{\text{load}}, \text{store}, \overline{\text{store}}, \text{cs-likely}\}$ is the alphabet, $\delta : Q \times \Sigma \mapsto Q$ is the state transition function, s is the start state, and e is the accepting, i.e., final state.

▶ **Definition 12.** *Given a PAG edge $n_1 \xrightarrow{\sigma} n_2$ with a corresponding state transition $\delta(q_1, \sigma) = q_2$, we define $(n_1, q_1) \rightarrowtail (n_2, q_2)$ as a one-step transition. The transitive closure of $\rightarrowtail$, denoted by $\rightarrowtail^+$, represents a multiple-step transition.*

We describe an antisymmetric property of our DFA in Lemmas 13 and 14 below.

▶ **Lemma 13.** *Let n_1 and n_2 be two PAG nodes. We have (1) $(n_1, s) \rightarrowtail^+ (n_2, \mathit{flows}) \implies (n_2, \overline{\mathit{flows}}) \rightarrowtail^+ (n_1, e)$ and (2) $(n_1, s) \rightarrowtail^+ (n_2, \overline{\mathit{flows}}) \implies (n_2, \mathit{flows}) \rightarrowtail^+ (n_1, e)$.*

Proof. To prove (1), we note that n_1 *flows* $n_2 \implies n_2\ \overline{\mathit{flows}}\ n_1$ in L_5. To prove (2), we note that n_1 *flows* $n \xrightarrow{\mathsf{store}\ |\ \overline{\mathsf{store}}} n_2 \implies n_2 \xrightarrow{\mathsf{store}\ |\ \mathsf{store}} n\ \overline{\mathit{flows}}\ n_1$ in L_5, where n is a PAG node. ◀

▶ **Lemma 14.** *Let n_1 and n_2 be two PAG nodes. We have $(n_2, \overline{\mathit{flows}}) \rightarrowtail^+ (n_1, e) \implies (n_1, s) \rightarrowtail^+ (n_2, \mathit{flows})$ and $(n_2, \mathit{flows}) \rightarrowtail^+ (n_1, e) \implies (n_1, s) \rightarrowtail^+ (n_2, \overline{\mathit{flows}})$.*

Proof. Proceeds as in the proof of Lemma 13 by noting [P-PARAM] given in Figure 9. ◀

In (18), we include a variable/object n in a method M (with its PAG denoted by G_M) into CI_{TURNER} if $L_5^{G_M}(P_1, P_2)^n$ does not hold for any two parameters P_1 and P_2 of M. In terms of our DFA, $L_5^{G_M}(P_1, P_2)^n$ holds iff $(P_1, s) \rightarrowtail^+ (n, q) \rightarrowtail^+ (P_2, e)$, where $q \in \{\mathit{flows}, \overline{\mathit{flows}}\}$.

The antisymmetric property of our DFA is exploited below.

▶ **Theorem 15.** *Let n be a variable/object in a method with P_1 and P_2 as its two parameters. $(P_1, s) \rightarrowtail^+ (n, q) \rightarrowtail^+ (P_2, e) \iff (P_2, s) \rightarrowtail^+ (n, \overline{q}) \rightarrowtail^+ (P_1, e)$, where $q \in \{\mathit{flows}, \overline{\mathit{flows}}\}$.*

Proof. Lemmas 13 and 14. ◀

As a result, we have designed an efficient algorithm for verifying $L_5^{G_M}(P_1, P_2)^n$ by verifying $n \in R_M(\mathit{flows}) \cap R_M(\overline{\mathit{flows}})$ for a method M (with G_M as its PAG), in which, $R : Q \mapsto \wp(\mathbb{V} \cup \mathbb{H})$ returns a set of nodes in G_M reached at a given state $q \in Q$ and $R^{-1} : \mathbb{V} \cup \mathbb{H} \mapsto \wp(Q)$ is the inverse of R. These two functions are computed according to the two rules given in Figure 11. The two rules are simple: [A-I] performs the initializations needed while [A-II] computes a fixed point for each function iteratively.

$$\frac{n \in N_M}{n \in R_M(s) \quad s \in R_M^{-1}(n)} \quad \text{[A-I]}$$

$$\frac{n_1 \xrightarrow{\sigma} n_2 \in E_M \quad q_1 \in R_M^{-1}(n_1) \quad \delta(q_1, \sigma) = q_2 \quad q_2 \notin R_M^{-1}(n_2)}{n_2 \in R_M(q_2) \quad q_2 \in R_M^{-1}(n_2)} \quad \text{[A-II]}$$

Figure 11 Rules for computing R_M and R_M^{-1} for a method M with $G_M = (N_M, E_M)$.

Given R_M computed above, we can now obtain CI_{TURNER} efficiently as follows:

$$CI_{\text{TURNER}} = \{n \mid M \in \mathbb{M}, n \text{ is a node in } G_M, n \notin R_M(\mathit{flows}) \cap R_M(\overline{\mathit{flows}})\} \qquad (19)$$

4.3 Time Complexity

The worst-case time complexity of TURNER in analyzing a program is linear in terms of its number of statements, for two reasons. First, $\mathsf{CI}_{\text{TURNER}}^{\text{OBS}}$ given in (3) and (4) can be found in $O(|\mathbb{H}|)$ based on the points-to information already computed by Andersen's analysis [1]. Second, R_M used in (19) for a method M, with its PAG denoted $G_M = (N_M, E_M)$, can be computed by the rules in Figure 11 in $O(|E_M| \times |Q|)$, where $|E_M|$ is the number of edges in G_M (constructed linearly based on the number of statements in M according to the rules in Figures 5 and 7–9) and $|Q|$, i.e., the number of states in the DFA (Figure 10), is 4.

5 Evaluation

We demonstrate that Turner can accelerate $k$obj significantly with only negligible precision loss, by being both substantially faster than Eagle [22] (the currently best precision-preserving pre-analysis) and substantially more precise than Zipper [19] (the currently best non-precision-preserving pre-analysis). We address the following three research questions:

- RQ1. Is Turner precise?
- RQ2. Is Turner efficient?
- RQ3. Is Turner effective (by exploiting object containment and reachability)?

We have implemented Turner in Soot [42], a program analysis and optimization framework for Java, on top of its context-insensitive Andersen's pointer analysis, Spark [17], and an object-sensitive version of Spark (i.e., $k$obj) developed by ourselves. Our pre-analysis is implemented in under 1000 lines of Java code, which will soon be released as an open-source tool at `http://www.cse.unsw.edu.au/~corg/turner`. To compare Turner with Eagle [22] and Zipper [19], we have implemented Eagle based on its three rules (in 600 lines of Java code) and used Zipper's latest version (b83b038).

As Zipper is evaluated in Doop [30], we have used an experimental setting that is as close as possible to its original one in several major aspects. First, objects such as `StringBuilder`, `StringBuffer` and `Throwable` objects are merged in terms of their dynamic types and then analyzed context-insensitively as is often done in Doop [6] and Wala [7]. Second, we perform an exception analysis together with $k$obj as in Doop by handling exception objects caught in terms of so-called exception-catch links [5]. Third, for type-filtering purposes performed on the elements of an array, we use the declared type of its elements instead of `java.lang.Object`. Finally, we use the summaries provided in Soot to handle native code.

We have carried out all the experiments on an Intel(R) Xeon(R) CPU E5-2637 3.5GHz machine with 512GB of RAM. We have selected a set of 12 popular Java programs, including 9 benchmarks from DaCapo2006 [3], and 3 Java applications (`checkstyle`, `JPC` and `findbugs`), which are commonly used in evaluating $k$obj [32, 40, 39, 13, 12]. The Java library used is `JRE1.6.0_45` (as the DaCapo2006 benchmarks rely only on an older version of JRE). We use Tamiflex [4], a dynamic reflection analysis tool, to resolve Java reflection as is often done in the pointer analysis literature [31, 32, 39, 19, 22, 21].

The time budget used for running each object-sensitive pointer analysis on a program is set as 24 hours. The analysis time of a program is an average of three runs.

Table 3 presents our main results. We compare Turner with Eagle and Zipper in terms of their efficiency and precision tradeoffs made on improving $k$obj. For each $k \in \{2, 3\}$ considered, $k$obj is the baseline, Z-$k$obj, E-$k$obj and T-$k$obj are the versions of $k$obj for performing selective context-sensitivity under Zipper, Eagle and Turner, respectively.

5.1 RQ1: Precision

Table 3 lists four common metrics used for measuring the precision of a context-sensitive pointer analysis [31, 41, 19, 22, 21] in terms of its context-insensitive points-to information obtained (as described in Section 2.1): (1) *#may-fail-casts*: 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) *#avg-pts*: the average number of objects pointed by a variable, i.e., the average points-to set size.

Eagle is designed to be precision-preserving by ensuring that E-$k$obj produces exactly the same context-insensitive points-to information as $k$obj. Thus, E-2obj and E-3obj

Table 3 Main results. For a given $k \in \{2, 3\}$, the speedups of E-kOBJ, Z-kOBJ, and T-kOBJ are normalized with kOBJ as the baseline. For all the metrics except "Speedup", smaller is better.

	Metrics	2OBJ	E-2OBJ	Z-2OBJ	T-2OBJ	3OBJ	E-3OBJ	Z-3OBJ	T-3OBJ
antlr	Time (s)	24.5	12.4	12.7	6.8	628.9	570.8	141.4	196.5
	Speedup	-	2.0x	1.9x	3.6x	-	1.1x	4.4x	3.2x
	#may-fail-casts	516	516	565	516	456	456	513	456
	#call-edges	50975	50975	51203	50975	50948	50948	51176	50948
	#poly-calls	1607	1607	1629	1607	1600	1600	1622	1600
	#avg-pts	6.110	6.110	6.585	6.125	4.927	4.927	5.427	4.945
bloat	Time (s)	412.6	290.9	324.2	138.9	10648.2	6994.7	6878.9	1902.8
	Speedup	-	1.4x	1.3x	3.0x	-	1.5x	1.5x	5.6x
	#may-fail-casts	1295	1295	1349	1295	1198	1198	1256	1198
	#call-edges	56488	56488	56988	56488	56258	56258	56837	56258
	#poly-calls	1549	1549	1587	1549	1535	1535	1577	1535
	#avg-pts	14.796	14.796	15.672	14.816	13.995	13.995	14.802	14.019
chart	Time (s)	206.2	107.5	28.3	75.1	*OoM*	12346.4	522.7	7886.1
	Speedup	-	1.9x	7.3x	2.7x	-	-	-	-
	#may-fail-casts	1339	1339	1410	1339	-	1239	1316	1239
	#call-edges	72426	72426	73009	72426	-	71987	72640	71987
	#poly-calls	1988	1988	2011	1988	-	1962	1989	1962
	#avg-pts	4.905	4.905	5.363	4.971	-	4.149	4.799	4.168
eclipse	Time (s)	10680.5	5885.3	4122.8	4686.0	*OoM*	*OoM*	*OoM*	*OoM*
	Speedup	-	1.8x	2.6x	2.3x	-	-	-	-
	#may-fail-casts	3551	3551	3718	3551	-	-	-	-
	#call-edges	162208	162208	163186	162208	-	-	-	-
	#poly-calls	9525	9525	9572	9525	-	-	-	-
	#avg-pts	17.334	17.334	19.691	17.519	-	-	-	-
fop	Time (s)	18.7	10.2	6.9	5.2	728.1	651.6	123.8	187.3
	Speedup	-	1.8x	2.7x	3.6x	-	1.1x	5.9x	3.9x
	#may-fail-casts	414	414	460	414	362	362	416	362
	#call-edges	34173	34173	34406	34173	34146	34146	34379	34146
	#poly-calls	816	816	841	816	809	809	834	809
	#avg-pts	3.577	3.577	4.132	3.597	3.359	3.359	3.942	3.383
luindex	Time (s)	15.7	9.4	6.3	4.6	596.3	532.6	131.7	185.0
	Speedup	-	1.7x	2.5x	3.4x	-	1.1x	4.5x	3.2x
	#may-fail-casts	402	402	455	402	348	348	405	348
	#call-edges	33449	33449	33689	33449	33422	33422	33662	33422
	#poly-calls	905	905	932	905	898	898	925	898
	#avg-pts	3.595	3.595	4.285	3.612	3.352	3.352	4.072	3.374
lusearch	Time (s)	22.3	15.8	11.1	10.4	1968.0	1736.8	523.5	881.1
	Speedup	-	1.4x	2.0x	2.1x	-	1.1x	3.8x	2.2x
	#may-fail-casts	417	417	473	417	366	366	425	366
	#call-edges	36247	36247	36485	36247	36220	36220	36458	36220
	#poly-calls	1103	1103	1131	1103	1096	1096	1124	1096
	#avg-pts	3.611	3.611	4.229	3.627	3.358	3.358	3.959	3.381
pmd	Time (s)	42.1	23.9	23.8	18.3	1504.0	1380.1	358.6	266.2
	Speedup	-	1.8x	1.8x	2.3x	-	1.1x	4.2x	5.7x
	#may-fail-casts	1174	1174	1252	1174	1116	1116	1199	1116
	#call-edges	59664	59664	59832	59664	59599	59599	59767	59599
	#poly-calls	2329	2329	2354	2329	2322	2322	2347	2322
	#avg-pts	4.943	4.943	6.378	4.954	4.684	4.684	5.973	4.698
xalan	Time (s)	243.2	121.8	54.2	90.9	25424.4	6771.9	694.2	1386.4
	Speedup	-	2.0x	4.5x	2.7x	-	3.8x	36.6x	18.3x
	#may-fail-casts	569	569	629	569	516	516	582	516
	#call-edges	45916	45916	46113	45916	45884	45884	46086	45884
	#poly-calls	1589	1589	1611	1589	1582	1582	1604	1582
	#avg-pts	4.253	4.253	5.258	4.272	4.096	4.096	5.014	4.119
checkstyle	Time (s)	1240.6	710.2	484.3	339.3	*OoM*	*OoM*	*OoM*	*OoM*
	Speedup	-	1.7x	2.6x	3.7x	-	-	-	-
	#may-fail-casts	1129	1129	1203	1129	-	-	-	-
	#call-edges	66702	66702	67528	66702	-	-	-	-
	#poly-calls	2188	2188	2246	2188	-	-	-	-
	#avg-pts	6.380	6.380	10.070	6.491	-	-	-	-
JPC	Time (s)	101.9	59.2	31.0	44.0	2371.1	1172.9	175.9	316.8
	Speedup	-	1.7x	3.3x	2.3x	-	2.0x	13.5x	7.5x
	#may-fail-casts	1364	1364	1438	1364	1209	1209	1281	1209
	#call-edges	81003	81003	81590	81003	79315	79315	79893	79315
	#poly-calls	4255	4255	4301	4255	4115	4115	4159	4115
	#avg-pts	5.050	5.050	5.486	5.065	4.434	4.434	4.752	4.458
findbugs	Time (s)	1820.6	681.1	128.7	150.9	*OoM*	*OoM*	2133.8	1947.0
	Speedup	-	2.7x	14.1x	12.1x	-	-	-	-
	#may-fail-casts	2037	2037	2100	2037	-	-	1884	1650
	#call-edges	87532	87532	88134	87532	-	-	87289	86599
	#poly-calls	3472	3472	3487	3472	-	-	3463	3441
	#avg-pts	8.011	8.011	8.804	8.058	-	-	7.203	6.632

Table 4 Context-sensitive facts (in millions). For all the metrics, smaller is better.

	Metrics	2obj	E-2obj	Z-2obj	T-2obj	3obj	E-3obj	Z-3obj	T-3obj
antlr	#cs-gpts	4.0K	3.8K	4.8K	2.2K	6.6K	6.0K	12.2K	2.8K
	#cs-pts	8.7M	4.9M	8.8M	1.5M	83.4M	63.4M	72.4M	33.3M
	#cs-fpts	0.4M	0.3M	0.4M	0.2M	10.2M	9.9M	10.3M	8.0M
	#cs-calls	2.4M	1.8M	1.0M	0.7M	38.5M	33.5M	6.8M	25.1M
	Total	11.5M	7.1M	10.2M	2.4M	132.1M	106.7M	89.6M	66.4M
bloat	#cs-gpts	3.2K	3.0K	4.0K	2.2K	5.1K	4.3K	11.3K	3.1K
	#cs-pts	120.4M	82.4M	111.1M	36.9M	1196.0M	856.5M	1137.5M	230.8M
	#cs-fpts	4.0M	4.0M	5.1M	3.7M	35.8M	35.4M	51.3M	30.6M
	#cs-calls	35.5M	32.1M	29.5M	15.0M	371.7M	340.5M	298.2M	109.9M
	Total	159.9M	118.4M	145.7M	55.6M	1603.6M	1232.5M	1487.0M	371.3M
chart	#cs-gpts	14.3K	13.0K	10.8K	8.2K	-	34.5K	26.3K	22.0K
	#cs-pts	64.3M	36.7M	17.0M	19.9M	-	1378.0M	171.2M	1005.7M
	#cs-fpts	1.5M	1.1M	0.8M	1.0M	-	55.4M	24.8M	48.8M
	#cs-calls	20.5M	12.2M	2.5M	8.7M	-	356.0M	23.9M	260.8M
	Total	86.4M	49.9M	20.4M	29.7M	-	1789.4M	220.0M	1315.3M
eclipse	#cs-gpts	40.6K	39.9K	28.8K	10.0K	-	-	-	-
	#cs-pts	991.9M	742.7M	744.5M	565.5M	-	-	-	-
	#cs-fpts	21.8M	21.4M	20.4M	16.2M	-	-	-	-
	#cs-calls	609.1M	342.7M	188.6M	296.5M	-	-	-	-
	Total	1622.8M	1106.8M	953.6M	878.2M	-	-	-	-
fop	#cs-gpts	3.1K	2.9K	3.7K	2.1K	4.5K	3.8K	9.8K	2.7K
	#cs-pts	3.7M	2.1M	3.6M	1.0M	70.3M	56.1M	48.8M	33.5M
	#cs-fpts	0.2M	0.2M	0.2M	0.2M	9.7M	9.4M	9.4M	7.9M
	#cs-calls	1.1M	0.9M	0.5M	0.5M	33.7M	29.8M	4.2M	25.0M
	Total	5.0M	3.2M	4.2M	1.6M	113.7M	95.3M	62.5M	66.4M
luindex	#cs-gpts	2.8K	2.6K	3.8K	1.9K	4.5K	3.9K	11.0K	2.7K
	#cs-pts	3.8M	2.2M	4.2M	1.1M	67.6M	54.2M	56.5M	33.2M
	#cs-fpts	0.2M	0.2M	0.2M	0.2M	9.7M	9.4M	10.8M	8.0M
	#cs-calls	1.1M	0.9M	0.5M	0.5M	33.1M	29.6M	4.7M	25.1M
	Total	5.2M	3.3M	4.9M	1.7M	110.4M	93.2M	72.0M	66.3M
lusearch	#cs-gpts	3.0K	2.7K	3.8K	1.9K	4.2K	3.5K	10.3K	2.5K
	#cs-pts	5.8M	3.9M	5.1M	2.2M	167.7M	151.6M	115.3M	92.2M
	#cs-fpts	0.3M	0.2M	0.2M	0.2M	11.2M	11.0M	11.0M	9.4M
	#cs-calls	2.3M	1.9M	1.0M	1.4M	108.1M	94.9M	40.5M	80.8M
	Total	8.4M	6.0M	6.4M	3.8M	287.1M	257.5M	166.9M	182.4M
pmd	#cs-gpts	3.9K	3.6K	5.9K	2.5K	5.6K	4.9K	23.8K	3.4K
	#cs-pts	12.2M	7.6M	15.1M	4.1M	144.6M	108.8M	184.5M	45.5M
	#cs-fpts	1.1M	1.0M	1.1M	0.9M	15.9M	15.3M	19.0M	11.7M
	#cs-calls	3.6M	2.6M	2.1M	1.7M	58.5M	49.0M	17.0M	33.3M
	Total	16.9M	11.1M	18.4M	6.7M	219.0M	173.1M	220.5M	90.6M
xalan	#cs-gpts	3.9K	3.6K	3.6K	2.4K	15.5K	13.5K	10.3K	6.1K
	#cs-pts	99.1M	45.9M	20.1M	14.3M	1795.3M	987.3M	253.0M	104.5M
	#cs-fpts	2.5M	2.4M	1.8M	1.9M	70.9M	63.6M	18.8M	27.0M
	#cs-calls	26.0M	19.3M	4.7M	17.2M	432.4M	300.8M	35.3M	168.1M
	Total	127.6M	67.6M	26.6M	33.3M	2298.6M	1351.7M	307.1M	299.6M
checkstyle	#cs-gpts	7.8K	7.5K	11.5K	3.9K	-	-	-	-
	#cs-pts	145.0M	107.2M	118.2M	38.0M	-	-	-	-
	#cs-fpts	2.5M	2.3M	3.0M	1.6M	-	-	-	-
	#cs-calls	78.6M	34.5M	23.2M	21.1M	-	-	-	-
	Total	226.1M	144.0M	144.4M	60.7M	-	-	-	-
JPC	#cs-gpts	7.9K	7.1K	7.7K	5.7K	22.1K	19.5K	17.5K	10.2K
	#cs-pts	28.7M	18.8M	13.9M	12.1M	618.1M	319.8M	68.6M	69.1M
	#cs-fpts	1.2M	0.9M	1.0M	0.9M	22.8M	20.0M	13.0M	13.0M
	#cs-calls	9.6M	7.1M	2.7M	5.8M	95.2M	61.4M	7.2M	38.4M
	Total	39.6M	26.9M	17.6M	18.8M	736.1M	401.3M	88.8M	120.5M
findbugs	#cs-gpts	33.5K	32.9K	10.7K	4.0K	-	-	45.6K	6.0K
	#cs-pts	326.4M	245.0M	57.2M	37.8M	-	-	545.9M	183.3M
	#cs-fpts	15.7M	15.5M	4.7M	1.1M	-	-	59.4M	26.6M
	#cs-calls	120.0M	58.3M	11.9M	9.6M	-	-	96.4M	138.5M
	Total	462.0M	318.9M	73.8M	48.5M	-	-	701.7M	348.5M

achieve trivially the same precision in all the four metrics. Zipper is designed to accelerate *k*obj heuristically as much as possible (by also ignoring the last two value-flow patterns in Figure 1) while allowing sometimes a significant loss of precision. For 2obj, Z-2obj has caused its #avg-pts to increase by 18.1% on average, resulting in the average percentage precision losses of 7.8%, 0.7%, and 1.7% for #may-fail-casts, #call-edges, and #poly-calls, respectively. For 3obj, Z-3obj has caused its #avg-pts to increase by 16.2% on average, resulting in the average percentage precision losses of 10.8%, 0.7%, and 2.0% for #may-fail-casts, #call-edges, and #poly-calls, respectively. In this paper, Turner is designed to trade only a slight loss of precision for efficiency (by reasoning all the four value-flow patterns in Figure 1 (implicitly) using a DFA based on object containment and reachability). Despite

Table 5 Times spent by SPARK and the three pre-analyses in seconds.

	antlr	bloat	chart	eclipse	fop	luindex	lusearch	pmd	xalan	checkstyle	JPC	findbugs	Avg
SPARK	9.0	10.7	17.2	38.6	8.1	7.4	7.9	13.5	9.5	16.8	19.3	19.8	14.8
EAGLE	3.5	3.8	9.9	34.6	2.8	2.7	3.0	9.3	6.1	9.2	9.6	12.1	8.9
ZIPPER	5.4	6.5	17.1	38.9	4.4	4.2	4.6	9.5	9.0	17.9	11.5	17.4	12.2
TURNER	0.8	0.9	1.4	2.4	0.5	0.5	0.5	1.1	0.8	1.2	1.2	1.3	1.1

some slightly imprecise points-to information produced (with #avg-pts increasing by 0.6% and 0.5% under T-2OBJ and T-3OBJ, respectively), both T-2OBJ and T-3OBJ preserve the precision for #may-fail-casts, #call-edges, and #poly-calls across all the 12 programs.

5.2 RQ2: Efficiency

On average, as shown in Table 3, T-kOBJ is faster than E-kOBJ but slower than Z-kOBJ. By adopting the context selections prescribed by each of the three pre-analyses, kOBJ runs faster under all the configurations. We compare TURNER with EAGLE and ZIPPER below.

- **T-kOBJ vs. E-kOBJ.** Both achieve the same precision for #may-fail-casts, #call-edges, and #poly-calls across the 12 benchmarks for $k \in \{2, 3\}$, but T-kOBJ is faster in each case. For $k = 2$, the speedups of T-2OBJ over 2OBJ range from 2.1x (for `lusearch`) to 12.1x (for `findbugs`) with an average of 3.6x. In contrast, the speedups of E-2OBJ over 2OBJ range from 1.4x (for `bloat` and `lusearch`) to 2.7x (for `findbugs`) with an average of 1.8x only. For $k = 3$, the speedups of T-3OBJ over 3OBJ range from 2.2x (for `lusearch`) to 18.3x (for `xalan`) with an average of 6.2x, while the speedups of E-3OBJ over 3OBJ range from 1.1x (for `antlr`, `fop`, `luindex`, `lusearch`, and `pmd`) to 3.8x (for `xalan`) with an average of 1.6x only. Thus, the speedups of T-kOBJ over E-kOBJ are 1.9x when $k = 2$ and 3.4x (with `chart` included even though 3OBJ is unscalable) when $k = 3$.
  In addition, T-kOBJ exhibits better scalability than E-kOBJ. For the four benchmarks, `chart`, `eclipse`, `checkstyle` and `findbugs`, that are unscalable under 3OBJ, T-3OBJ can now analyze `chart` and `findbugs` successfully but E-3OBJ can analyze `chart` only.
- **T-kOBJ vs. Z-kOBJ.** Despite its substantially better precision, T-kOBJ is faster in seven programs when $k = 2$ and three when $k = 3$. Compared with the kOBJ baseline, the average speedups achieved by T-kOBJ and Z-kOBJ are 3.6x and 3.9x, respectively, when $k = 2$, and 6.2x and 9.3x, respectively, when $k = 3$. As a result, Z-kOBJ is faster than T-kOBJ by 1.1x when $k = 2$ and 2.7x (with `chart` and `findbugs` included) when $k = 3$, on average. In terms of scalability, T-kOBJ is on par with Z-kOBJ for $k \in \{2, 3\}$.

Table 4 gives the numbers of context-sensitive facts established by kOBJ, E-kOBJ, Z-kOBJ and T-kOBJ, with #cs-gpts, #cs-pts and #cs-fpts representing the numbers of context-sensitive objects pointed by global variables (i.e., static fields), local variables and instance fields, respectively, and #cs-calls representing 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, Z-3OBJ is significantly faster than T-3OBJ and E-3OBJ for `chart` as its number of context-sensitive facts is significantly less than the other two. Similarly, T-3OBJ is also much faster than E-3OBJ and Z-3OBJ for `bloat`. However, the analysis time of a pointer analysis is not linearly proportional to the number of context-sensitive facts computed [41]. For example, T-3OBJ is faster than 3OBJ by 3.2x for `antlr` but achieves a percentage time reduction of only 49.7%.

Table 5 gives the times spent by SPARK [17] (an implementation of context-insensitive Andersen's analysis [1]) and the three pre-analyses, EAGLE, ZIPPER and TURNER. As discussed earlier, each pre-analysis relies on the points-to information computed by SPARK to make its context selection decisions. TURNER is significantly faster than EAGLE and ZIPPER across all the 12 programs. On average, we have 1.1 seconds (TURNER), 8.9 seconds (EAGLE) and 12.2 seconds (ZIPPER). EAGLE is a single-threaded pre-analysis, ZIPPER is multi-threaded (with 16 threads used in our experiments), TURNER is currently single-threaded but is embarrassingly parallel, as it is intra-procedural. Without any parallelization, TURNER exhibits already negligible analysis times as it runs linearly in terms of the number of statements in a program.

5.3 RQ3: Effectiveness

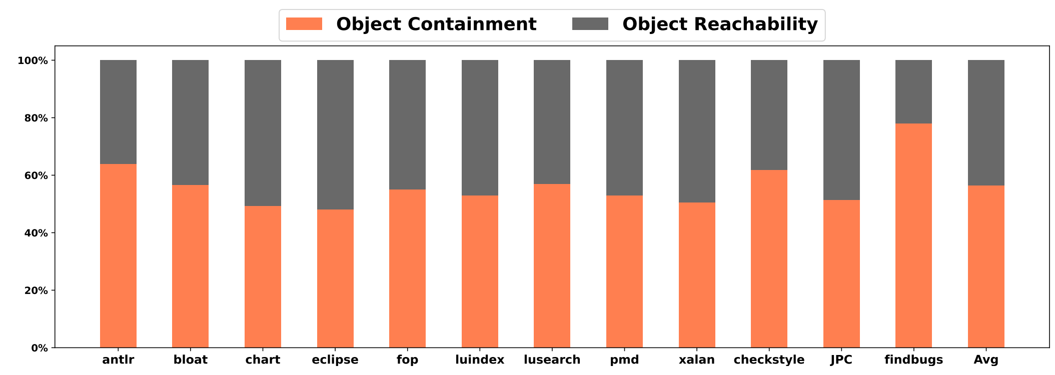


Figure 12 Percentage contributions made by TURNER's two analysis stages for the speedups of T-2OBJ over 2OBJ.

TURNER relies on object containment and reachability to make its context selections. In order to understand roughly their percentage contributions to the speedups achieved by T-kOBJ over kOBJ, let us consider two versions of T-kOBJ: (1) T-kOBJ^C, where only object containment is exploited, i.e., the objects in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ are context-insensitive and all the rest (the variables/objects in $(\mathbb{V} \cup \mathbb{G} \cup \mathbb{H}) \setminus \mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$) are handled as in kOBJ, and (2) T-kOBJ^R, where only object reachability is exploited by assuming $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} = \varnothing$. Let T-$kOBJ^S_{\text{Speedup}}$ be the speedup obtained by T-kOBJ^S over kOBJ, where $S \in \{C, R, \epsilon\}$, for a program. Certainly, T-kOBJ$^C_{\text{Speedup}}$ + T-kOBJ$^R_{\text{Speedup}}$ = T-kOBJ$_{\text{Speedup}}$ is not expected for a program, as the common contribution made by T-kOBJ^C and T-kOBJ^R towards T-kOBJ$_{\text{Speedup}}$ cannot be meaningfully isolated. Instead, we consider T-kOBJ$^S_{\text{Speedup}}$/(T-kOBJ$^C_{\text{Speedup}}$ + T-kOBJ$^R_{\text{Speedup}}$), where $S \in \{C, R\}$, as the relative percentage contribution made by T-kOBJ^S towards T-kOBJ$_{\text{Speedup}}$ in order to gain a rough understanding about whether both stages are indispensable. Figure 12 illustrates the case for accelerating 2OBJ by T-2OBJ, demonstrating that both object containment and object reachability are indeed exploited beneficially for real-world programs.

Our work is largely driven by our insight stated in Observation 3. Therefore, TURNER is designed to exploit both object containment and reachability to classify the objects, and consequently, the variables in a program as context-sensitive or context-insensitive.

Figure 13 gives a Venn diagram showing how TURNER classifies the containers, i.e., objects in a program. Based on object containment (Observation 3), $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} = \mathsf{TopCon} \cup \mathsf{BotCon}$ gives the set of precision-uncritical, i.e., context-insensitive objects identified. Based on object reachability (performed by our DFA), $CI^{DFA}_{\text{TURNER}} \subseteq \mathbb{H} \setminus \mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ gives an additional set of context-insensitive sets identified. Thus, $CS_{\text{TURNER}} = \mathbb{H} \setminus (\mathsf{CI}^{\text{OBS}}_{\text{TURNER}} \cup CI^{DFA}_{\text{TURNER}})$ represents the set of context-sensitive objects identified. On average, across the 12 programs evaluated,

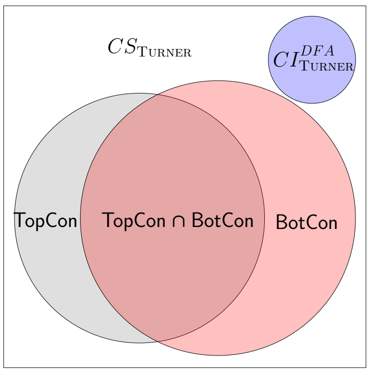


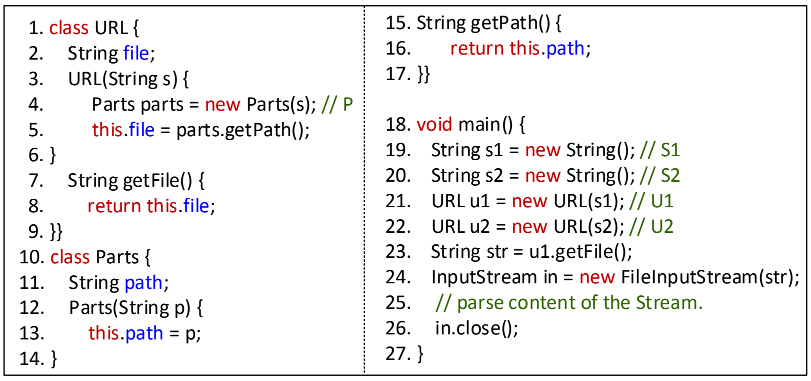
Figure 13 The Venn diagram of the objects in a program.

the ratios of $|\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}|$, $|CI^{DFA}_{\text{TURNER}}|$ and $|CS_{\text{TURNER}}|$ over $|\mathbb{H}|$ are 61.3%, 4.9%, and 33.8%, respectively. As the performance benefits of making different objects context-insensitive can be drastically different (which are hard to measure individually), these ratios, together with Figure 12, demonstrate again the effectiveness of TURNER's two analysis stages.

Finally, we give two examples abstracted from the JDK library to explain why TURNER does not lose any precision in #call-edges, #may-fail-casts, and #poly-calls even though it suffers from a small loss of precision in #avg-pts across the 12 programs evaluated. TURNER can render some points-to sets imprecise when some top/bottom containers that are classified as precision-uncritical in $\mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ should have been analyzed context-sensitively.

Figure 14 illustrates a case in which whether the object `P` created in line 4 (a top container according to Observation 3) is analyzed context-sensitively or not affects $\overline{\texttt{pts}}(\texttt{str})$ obtained in line 23. Consider 2OBJ, which will analyze `P` context-sensitively. When analyzing lines 19–22, we find that $\texttt{pts}(\texttt{u}i, [\,]) = \{(\texttt{U}i, [\,])\} \wedge \texttt{pts}(\texttt{U}i\texttt{.file}, [\,]) = \texttt{pts}(\texttt{P.path}, [\texttt{U}i]) = \{(\texttt{S}i, [\,])\}$, where $1 \leqslant i \leqslant 2$. When analyzing line 23, we find that $\texttt{pts}(\texttt{str}, [\,]) = \{(\texttt{S1}, [\,])\}$. Context-insensitively, 2OBJ thus obtains $\overline{\texttt{pts}}(\texttt{str}) = \{\texttt{S1}\}$. In the case of T-2OBJ, however, $\texttt{P} \in \mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ will be analyzed context-insensitively instead. When analyzing lines 19–22, we have $\texttt{pts}(\texttt{u}i, [\,]) = \{(\texttt{U}i, [\,])\} \wedge \texttt{pts}(\texttt{U}i\texttt{.file}, [\,]) = \texttt{pts}(\texttt{P.path}, [\,]) = \{(\texttt{S1}, [\,]), (\texttt{S2}, [\,])\}$, where $1 \leqslant i \leqslant 2$. As `P` is context-insensitive, analyzing line 23 this time will give rise to $\texttt{pts}(\texttt{str}, [\,]) = \{(\texttt{S1}, [\,]), (\texttt{S2}, [\,])\}$. Thus, context-insensitively, T-2OBJ obtains $\overline{\texttt{pts}}(\texttt{str}) = \{\texttt{S1}, \texttt{S2}\}$, which contains a spurious target `S2` introduced for `str`. Despite this loss of precision in #avg-pts, however, T-2OBJ does not lose any precision in #may-fail-casts, #call-edges, and #poly-calls, as both `S1` and `S2` have exactly the same type, `java.lang.String`.

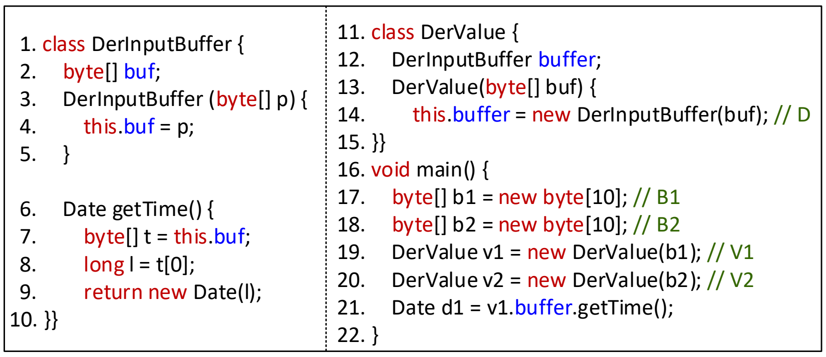
Figure 15 illustrates another case in which whether the object `D` created in line 14 (a bottom container according to Observation 3) is analyzed context-sensitively or not affects $\overline{\texttt{pts}}(\texttt{t})$ obtained in line 7. Consider 2OBJ, which will analyze `D` context-sensitively. When analyzing lines 17–20, we find that $\texttt{pts}(\texttt{v}i, [\,]) = \{(\texttt{V}i, [\,])\} \wedge \texttt{pts}(\texttt{V}i\texttt{.buffer}, [\,]) = \{(\texttt{D}, [\texttt{V}i])\} \wedge \texttt{pts}(\texttt{D.buf}, [\texttt{V}i]) = \{(\texttt{B}i, [\,])\}$, where $1 \leqslant i \leqslant 2$. When analyzing line 7, we find that $\texttt{pts}(\texttt{t}, [\texttt{D}, \texttt{V1}]) = \{(\texttt{B1}, [\,])\}$. Context-insensitively, 2OBJ thus obtains $\overline{\texttt{pts}}(\texttt{t}) = \{\texttt{B1}\}$. In the case of T-2OBJ, however, $\texttt{D} \in \mathsf{CI}^{\text{OBS}}_{\text{TURNER}}$ will be analyzed context-insensitively instead. When analyzing lines 17–20, we have $\texttt{pts}(\texttt{v}i, [\,]) = \{(\texttt{V}i, [\,])\} \wedge \texttt{pts}(\texttt{V}i\texttt{.buffer}, [\,]) = \{(\texttt{D}, [\,])\} \wedge \texttt{pts}(\texttt{D.buf}, [\,]) = \{(\texttt{B}i, [\,])\}$, where $1 \leqslant i \leqslant 2$. As `t` is context-insensitive, analyzing line 7 will give rise to $\texttt{pts}(\texttt{t}, [\,]) = \{(\texttt{B1}, [\,]), (\texttt{B2}, [\,])\}$. Thus, context-insensitively, T-2OBJ obtains $\overline{\texttt{pts}}(\texttt{t}) = \{\texttt{B1}, \texttt{B2}\}$, which contains a spurious target `B2` introduced for `t`. Despite this loss of precision in #avg-pts, T-2OBJ loses no precision in #may-fail-casts, #call-edges, and #poly-calls, as both `B1` and `B2` have exactly the same type, `java.lang.byte[]`, and in



```
1. class URL {
2.    String file;
3.    URL(String s) {
4.       Parts parts = new Parts(s); // P
5.       this.file = parts.getPath();
6. }
7.    String getFile() {
8.       return this.file;
9. }}
10. class Parts {
11.    String path;
12.    Parts(String p) {
13.       this.path = p;
14. }
15. String getPath() {
16.       return this.path;
17. }}

18. void main() {
19.    String s1 = new String(); // S1
20.    String s2 = new String(); // S2
21.    URL u1 = new URL(s1); // U1
22.    URL u2 = new URL(s2); // U2
23.    String str = u1.getFile();
24.    InputStream in = new FileInputStream(str);
25.     // parse content of the Stream.
26.     in.close();
27. }
```

Figure 14 Imprecise points-to information computed by T-2OBJ for a top container P.



```
1. class DerInputBuffer {
2.    byte[] buf;
3.    DerInputBuffer (byte[] p) {
4.       this.buf = p;
5.    }

6.    Date getTime() {
7.       byte[] t = this.buf;
8.       long l = t[0];
9.       return new Date(l);
10. }}
11. class DerValue {
12.    DerInputBuffer buffer;
13.    DerValue(byte[] buf) {
14.       this.buffer = new DerInputBuffer(buf); // D
15. }}
16. void main() {
17.    byte[] b1 = new byte[10]; // B1
18.    byte[] b2 = new byte[10]; // B2
19.    DerValue v1 = new DerValue(b1); // V1
20.    DerValue v2 = new DerValue(b2); // V2
21.    Date d1 = v1.buffer.getTime();
22. }
```

Figure 15 Imprecise points-to information computed by T-2OBJ for a bottom container D.

addition, each array object pointed by `t` is used in line 8 for obtaining a long integer only.

6 Related Work

There are two approaches for developing pre-analyses for improving the efficiency and scalability of object-sensitive pointer analysis (kOBJ) for Java: the precision-preserving approach [22, 21] and non-precision-preserving approach [32, 19, 13, 9]. EAGLE [22, 21] aims to improve the efficiency of kOBJ while preserving its precision by reasoning about all the four value-flow patterns in Figure 1 implicitly via CFL reachability to make its context selections conservatively, thereby limiting the speedups achieved. In this paper, TURNER addresses its limitation by trading a slight loss of precision for greater performance speedups. On the other hand, ZIPPER [19], as a representative non-precision-preserving pre-analysis [32, 19, 13, 9], aims to trade precision for efficiency by examining the first two value-flow patterns in Figure 1 heuristically to make its context selections, achieving sometimes greater speedups than EAGLE but at a substantial loss of precision for some programs. In this paper, TURNER addresses its limitation by trading a slight loss of efficiency for greater precision by exploiting object containment (Observation 3) and then reasoning about all the four value-flow patterns in Figure 1 implicitly via an intra-procedural object reachability analysis.

There are other types of pre-analyses for kOBJ. MAHJONG [39] sacrifices the precision of alias analysis (by merging objects of the same dynamic type) in order to improve the efficiency of kOBJ at a small loss of precision for a class of so-called type-dependent clients, such as call graph construction, may-fail casting, and polymorphic call detection. In contrast, Turner is designed to be a general-purpose pointer analysis to support all possible applications that rely on points-to information, including not only type-dependent clients but also alias analysis. Jeong et al. [13] apply machine learning to select the lengths of calling contexts for different methods analyzed by $k$obj for a particular client (e.g., may-fail-casting). In contrast, Turner makes its context selections by exploiting object containment and reachability.

There are also research efforts for developing pre-analyses for other programming languages. For example, Wei and Ryder [43] present an adaptive context-sensitive analysis for JavaScript. They extract user-specific function characteristics from an inexpensive pre-analysis and then apply a decision-tree-based machine learning technique to correlate these features with different types of context-sensitivity, e.g., 1-callsite, 1-object and i-th-parameter, achieving better precision and efficiency than any single context-sensitive analysis evaluated.

Elsewhere [14, 40, 12], pre-analyses are also applied to improve the precision of $k$obj at the expense of its efficiency. This thread of research is orthogonal to ours considered here.

There are other types of approaches for conducting pointer analysis in Java programs. Thiessen and Lhoták [41] propose to use context transformations rather than context strings as a new context abstraction for $k$obj, making it theoretically possible for $k$obj to run more efficiently with better precision. Instead of solving $k$obj as a whole-program analysis [17, 44, 23, 6, 18] as in this paper, demand-driven pointer analyses [37, 34, 45, 28, 38, 33] typically compute the points-to information for particular variables of interest, with call-site-sensitivity instead of object-sensitivity being often used.

Finally, Mohri and Nederhof [24] introduce an approach for over-approximating a context-free grammar (CFG) by a non-deterministic finite automaton (NFA). Prasanna *et al.* [16] adopt this approach to compute the liveness information required by a garbage collector for functional programs. For object-oriented pointer analysis, however, this is the first paper introducing an intra-procedural pre-analysis for determining selective context-sensitivity, based on a DFA over-approximated from a CFG that defines pointer analysis inter-procedurally.

7 Conclusion

We have introduced Turner, a simple, lightweight yet effective pre-analysis technique that can accelerate object-sensitive pointer analysis for Java programs with negligible precision loss. We exploit a key insight that many precision-uncritical objects in a program can be identified based on a pre-computed object containment relationship. Leveraging this approximation, we can reason about object reachability intra-procedurally to determine whether the remaining objects, together with all the variables, in the program are precision-critical or not. As a result, we have obtained a novel pre-analysis that can improve the efficiency of object-sensitive pointer analysis significantly while suffering only a small loss of precision in the points-to information produced. In particular, Turner is shown to preserve the precision of object-sensitive pointer analysis for three important clients, call graph construction, may-fail casting, and polymorphic call detection over a set of 12 popular Java programs evaluated.

We see several directions to move forward. First, we can incorporate the object allocation relationship (exploited earlier [40]) into our framework to mitigate some precision loss incurred in the scenarios shown in Figures 14 and 15. Second, we can sharpen the precision of $\mathsf{CI}^{\mathrm{OBS}}_{\mathrm{Turner}}$ with a more precise yet faster algorithm than Anderson's analysis [1]. Finally, we can analyze a method based on the context selections made earlier by exploiting a precision/efficiency tradeoff made possible by the modularity of our intra-procedural pre-analysis.

References

1 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

2 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick 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*, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2666356.2594299`.

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. The DaCapobenchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. Association for Computing Machinery. `doi:10.1145/1167515.1167488`.

4 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira 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*, pages 241–250, Honolulu, HI, USA, 2011. IEEE. `doi:10.1145/1985793.1985827`.

5 Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery. `doi:10.1145/1572272.1572274`.

6 Martin Bravenboer and Yannis 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*, pages 243–262, New York, NY, USA, 2009. Association for Computing Machinery. `doi:10.1145/1639949.1640108`.

7 IBM T.J. Watson Research Center. WALA: T.J. Watson Libraries for Analysis, 2020. URL: `http://wala.sourceforge.net/`.

8 Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–34, 2008. `doi:10.1145/1348250.1348255`.

9 Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi 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*, page 13–18, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3088515.3088519`.

10 Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling 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)*, pages 267–279, San Diego, CA, USA, 2019. IEEE. `doi:10.1109/ASE.2019.00034`.

11 Dongjie He, Lian Li, Lei Wang, Hengjie Zheng, Guangwei Li, and Jingling Xue. Understanding and detecting evolution-induced compatibility issues in Android apps. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 167–177, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3238147.3238185`.

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

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

14 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 423–434, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2499370.2462191`.

15 John Kodumal and Alex Aiken. The set constraint/cfl reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 207–218, New York, NY, USA, 2004. ACM. `doi:10.1145/996893.996867`.

16 Prasanna Kumar K., Amitabha Sanyal, and Amey Karkare. Liveness-based garbage collection for lazy languages. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, page 122–133, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2926697.2926698`.

17 Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using spark. In *International Conference on Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. `doi:10.5555/1765931.1765948`.

18 Lian Li, Cristina Cifuentes, and Nathan 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*, pages 343–353, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/2025113.2025160`.

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

20 Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming*, pages 15:1–15:27, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECOOP.2016.15`.

21 Jingbo Lu, Dongjie He, and Jingling 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. To appear.

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

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

24 Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation. In Jean-Claude Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, pages 153–163. Springer Netherlands, Dordrecht, 2001. `doi:10.1007/978-94-015-9719-7_6`.

25 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. Association for Computing Machinery. `doi:10.1145/1133255.1134018`.

26 Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998. `doi:10.1016/S0950-5849(98)00093-7`.

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

28 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and*

Optimization, pages 264–274, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2259016.2259050.

29 Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis.* New York University. Courant Institute of Mathematical Sciences , 1978.

30 Yannis Smaragdakis. Doop-framework for Java pointer and taint analysis (using p/taint), 2021. URL: https://bitbucket.org/yanniss/doop/.

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

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

33 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for Java. In *30th European Conference on Object-Oriented Programming*, pages 22:1–22:26, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ECOOP.2016.22.

34 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 387–400, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1133255.1134027.

35 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J Fink, and Eran Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 196–232. Springer, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-36946-9_8.

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

37 Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav 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*, page 59–76, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1103845.1094817.

38 Yulei Sui and Jingling 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*, pages 460–473, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2950290.2950296.

39 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*, pages 278–291, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3140587.3062360.

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

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

42 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., USA, 2010. doi:10.5555/781995.782008.

43 Shiyi Wei and Barbara G Ryder. Adaptive context-sensitive analysis for JavaScript. In *29th European Conference on Object-Oriented Programming*, pages 712–734, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.ECOOP.2015.712`.

44 John Whaley and Monica 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*, pages 131–144, New York, NY, USA, 2004. Association for Computing Machinery. `doi:10.1145/996841.996859`.

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