EAGLE: CFL-Reachability-Based Precision-Preserving Acceleration of Object-Sensitive Pointer Analysis with Partial Context Sensitivity

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

Object sensitivity is widely used as a context abstraction for computing the points-to information context-sensitively for object-oriented programming languages such as Java. Due to the combinatorial explosion of contexts in large object-oriented programs, k-object-sensitive pointer analysis (under k-limiting), denoted k-*obj*, is often inefficient even when it is scalable for small values of k, where $k \leqslant 2$ holds typically. A recent popular approach for accelerating k-*obj* trades precision for efficiency by instructing k-*obj* to analyze only some methods in a program context-sensitively, determined heuristically by a pre-analysis. In this article, we investigate how to develop a fundamentally different approach, EAGLE, for designing a pre-analysis that can make k-*obj* run significantly faster while maintaining its precision. The novelty of EAGLE is to enable k-*obj* to analyze a method with partial context sensitivity (i.e., context-sensitively for only some of its selected variables/allocation sites) by solving a context-free-language (CFL) reachability problem based on a new CFL-reachability formulation of k-*obj*. By regularizing one CFL for specifying field accesses and using another CFL for specifying method calls, we have formulated EAGLE as a fully context-sensitive taint analysis (without k-limiting) that is both effective (by selecting the variables/allocation sites to be analyzed by k-*obj* context-insensitively so as to reduce the number of context-sensitive facts inferred by k-*obj* in the program) and efficient (by running linearly in terms of the number of pointer assignment edges in the program). As EAGLE represents the first precision-preserving pre-analysis, our evaluation focuses on demonstrating its significant performance benefits in accelerating k-*obj* for a set of popular Java benchmarks and applications, with call graph construction, may-fail-casting, and polymorphic call detection as three important client analyses.

CCS Concepts: • **Theory of computation → Program analysis**;

Additional Key Words and Phrases: Pointer analysis, object sensitivity, CFL-reachability

ACM Reference format:
Jingbo Lu, Dongjie He, and Jingling Xue. 2021. EAGLE: CFL-Reachability-Based Precision-Preserving Acceleration of Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 46 (July 2021), 46 pages.
https://doi.org/10.1145/3450492

This work has been supported by Australian Research Council Grants (DP170103956 and DP180104069).
Authors' address: J. Lu, D. He, and J. Xue, School of Computer Science and Engineering, UNSW Sydney, Kensington, NSW 2052, Australia; emails: {jlu, dongjieh}@cse.unsw.edu.au, j.xue@unsw.edu.au.

1049-331X/2021/07-ART46 $15.00
https://doi.org/10.1145/3450492

1 INTRODUCTION

Pointer analysis computes statically the set of abstract objects that a pointer may point to in a program. It is widely recognized as the foundation for almost all other forms of static analysis adopted in a variety of areas such as security analysis, bug detection, program verification, compiler optimization, and software engineering.

For object-oriented languages such as Java, context sensitivity is known to provide highly useful precision for pointer analysis [16, 35]. A context-insensitive pointer analysis, such as the analysis of Andersen [1], analyzes a method without distinguishing its calling contexts, producing one points-to set per variable and one abstract object per allocation site in the method. In contrast, its context-sensitive counterpart analyzes a method separately under different calling contexts that abstract its different runtime invocations, thereby producing multiple points-to sets per variable (with one per context) and multiple abstract objects per allocation site (with one per context) in the method.

To tame the combinatorial explosion of calling contexts encountered in performing a context-sensitive pointer analysis, a context is usually represented by a sequence of k context elements, under k-limiting. There are two representative abstractions for object-oriented programs: (1) *k-callsite sensitivity* [34], which distinguishes the contexts of a method by its k-most-recent callsites, and (2) *k-object sensitivity* [26, 27], which distinguishes the contexts of a method by its receiver's k-most-recent allocation sites. In either case, a pointer analysis is said to be *fully context-sensitive* if $k = \infty$ (i.e., without k-limiting). In the past decade or so, object sensitivity has emerged as a better abstraction than callsite sensitivity in achieving precision and efficiency [12, 16, 35, 40].

However, even with k-limiting, k-object-sensitive pointer analysis, denoted *k-obj*, can be inefficient even when it scales under $k \leqslant 2$ [11, 12, 35, 36, 39, 40]. For a reasonably large object-oriented program, blindly applying the same degree of context sensitivity to each of its methods still causes the exponential blow-up in the number of contexts handled.

A recent popular approach for accelerating *k-obj* trades precision for efficiency [9, 12, 17, 36] by instructing *k-obj* to restrict context sensitivity to only some selected methods in a Java program. To make such selections, a pre-analysis is performed to determine heuristically whether a method should be analyzed by *k-obj* context-sensitively or context-insensitively *in its entirety*. Different heuristics are used, including client-specific machine learning techniques [12] (guided by improving the precision of a given client, e.g., may-fail-casting) and general-purpose techniques, such as user-supplied hints [9, 36] and pattern matching [17]. In general, developing such a heuristics-based pre-analysis entails experimenting with different heuristics to enable *k-obj* to achieve different efficiency/precision tradeoffs, since it will be difficult to quantify the effects of these heuristics on any precision-related metric in advance.

In this article, we introduce a fundamentally different approach, Eagle, to developing a pre-analysis that can scale *k-obj* for Java programs. As in many existing pre-analyses [9, 17, 36], our pre-analysis (which is presented in Algorithm 1) is also developed based on the pre-computed points-to information by the algorithm of Andersen [1]. Instead of accelerating *k-obj* at some loss of precision as in prior work [9, 12, 17, 36], Eagle preserves its precision (by computing the same points-to information) while making it run significantly faster than before. The novelty of Eagle is to enable *k-obj* to analyze a method with partial context sensitivity (i.e., context-sensitively only for some of its selected variables/allocation sites) by solving a **context-free-language (CFL)**-reachability problem based on a new CFL-reachability formulation of *k-obj*. By regularizing one CFL for specifying field accesses and using another CFL for specifying method calls, we have formulated Eagle as a fully context-sensitive taint analysis (without k-limiting) that is both effective (by selecting the variables/allocation sites to be analyzed by *k-obj* context-insensitively so as to

reduce the number of context-sensitive facts inferred by *k-obj* in the program) and efficient (by running linearly in terms of the number of pointer assignment edges in the program).

To the best of our knowledge, this is the first study on developing a pre-analysis to make *k-obj* run faster without losing precision, which is of both theoretical interest and practical significance. In theory, we believe that our new CFL-reachability formulation of *k-obj* represents a useful framework for developing precision-preserving pre-analyses and other transformations for improving the efficiency of *k-obj* (despite some challenges discussed in Section 2). Without exploiting the CFL-reachability information, we cannot even apply *k-obj* as a pre-analysis to achieve a precision-preserving acceleration of *k-obj*, since the points-to information computed by *k-obj* is not adequate enough in determining which variables/allocation sites should be analyzed context-sensitively. In practice, retaining the precision of *k-obj* can often be desirable. Avoiding even a slight loss of precision in *k-obj* may cause an underlying bug/security analysis tool to avoid generating some false positives that would have costed an overwhelming amount of post-processing time by the tool user.

In summary, this work makes the following major contributions:

- We present a new approach, Eagle, to scaling *k*-object-sensitive pointer analysis (*k-obj*) while preserving its precision for large Java programs by exploiting partial context sensitivity for the first time.
- We give a new CFL-reachability formulation of *k-obj*, which is instrumental in guiding us to develop our CFL-reachability-based pre-analysis (and possibly other transformations in the same framework in future work).
- We introduce a lightweight CFL-reachability-based pre-analysis for achieving a precision-preserving acceleration of *k-obj* with partial context sensitivity. Our pre-analysis is fully context-sensitive to maximize its effectiveness (i.e., reduce the number of context-sensitive facts inferred by *k-obj*) yet fast (as it is formulated as a taint analysis that is solvable linearly in terms of the pointer assignment edges in the program).
- We have implemented Eagle and all related analyses in Soot [43], a popular program analysis and optimization framework for Java and Android programs, on top of its context-insensitive Andersen's pointer analysis, Spark [15], and its object-sensitive version, Obj-Sens-Soot from MIT [8]. Note that both Spark and Obj-Sens-Soot are standard platforms for developing pointer analysis algorithms for analyzing real-world Java and Android programs in Soot. As Eagle represents the first precision-preserving pre-analysis for accelerating *k-obj*, we focus on demonstrating its significant performance benefits in accelerating *k-obj* by using a set of 12 popular Java benchmarks and applications, with call graph construction, may-fail-casting, and polymorphic call detection as three important client analyses, under a time budget of 24 hours. Eagle enables *k-obj* to run significantly faster for all programs. For each $k \in \{1, 2, 3\}$, *k-obj* can now scalably analyze no fewer programs than before: 12 in both cases ($k = 1$), an increase of 11 to 12 ($k = 2$), and an increase of 5 to 9 ($k = 3$). For the programs that are analyzable originally by *k-obj*, totaling 12 ($k = 1$), 11 ($k = 2$), and 5 ($k = 3$), *k-obj* (assisted by Eagle) exhibits increasingly better scalability, yielding the average speedups of 3.8× ($k = 1$), 5.2× ($k = 2$), and 6.6× ($k = 3$). For the four programs that were unanalyzable before (under 24 hours each) but analyzable now ($k = 3$), *k-obj* (assisted by Eagle) can now analyze all of them in just under 9.5 hours. Eagle has been open sourced [7].

The rest of this article is organized as follows. Section 2 motivates Eagle by discussing our key insights and some challenges faced in its development. Section 3 formulates our CFL-reachability-based pre-analysis for enabling precision-preserving partial context sensitivity in *k-obj*. Section 4

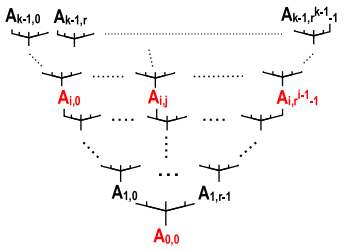
```
foo_{i,0}() {
  a = new A(); //A_{i,0}
  a.foo_{i-1,0}(); }
...
foo_{i,j}() {
  a = new A(); //A_{i,j}
  a.foo_{i-1,j÷r}(); }
...
foo_{i,r^i-1}() {
  a = new A(); //A_{i,r^i-1}
  a.foo_{i-1,r^{i-1}-1}(); }
```

```
foo_{0,0}() {
  a = new A(); //A_{0,0}
  b = new B();
  b.goo(a); }

goo(A p) {
  p.bar(...); }

bar(...) {
  z = ...;
  ... = new X(); //X
  ... }
```

(a) Code, where $\texttt{foo}_{i,j}()$ calls $\texttt{foo}_{i-1,j\div r}()$ $(0 < i < k)$



(b) Contexts for `bar()` from `p.bar()`

Fig. 1. *k-obj*: Object sensitivity and the combinatorial explosion of calling contexts in a large program.

evaluates the speedups achieved by Eagle in accelerating *k-obj*. Section 5 discusses the related work. Finally, Section 6 concludes the article.

2 MOTIVATION

We motivate our Eagle approach in accelerating *k-obj* while also preserving its precision for Java programs. In Section 2.1, we review object sensitivity as a context abstraction in *k-obj*, by highlighting the necessity for addressing *k-obj*'s scalability issue. In Section 2.2, we introduce our precision-preserving approach for accelerating *k-obj* with partial context sensitivity, based on a new CFL-reachability-based pre-analysis. As *k-obj* is context-sensitive but flow-insensitive, our pre-analysis relies on a new CFL-reachability formulation of *k-obj* by reasoning about the *value flow* (i.e., the flow of objects across the program in terms of data dependences). Thus, we will focus on describing our key insights and some challenges faced for achieving partial context sensitivity, assisted by small code examples (gradually modified to ease understanding), while leaving its actual CFL-reachability formulation in Section 3.

2.1 *k-obj*: Object-Sensitive Context Sensitivity

Object-sensitive pointer analysis [27] separates a method's calling contexts by using its receiver objects. Let each allocation site be identified by one (abstract) object allocated therein. In *k-obj*, an object o_0 is modeled context-sensitively by a so-called *heap context* of length $k-1$, $[o_{k-1}, \ldots, o_1]$, where o_i is the receiver object of a method in which o_{i-1} is allocated, where $0 < i < k$. As a result, a method with o_0 as its receiver will be analyzed context-sensitively multiple times, once for each of o_0's heap contexts $[o_{k-1}, \ldots, o_1]$, under a so-called *method context* $[o_{k-1}, \ldots, o_0]$ of length k. Unlike callsite sensitivity [34], object sensitivity can thus be specified by either heap or method contexts. Thus, the points-to set of a variable v in a method m, $pt(c, v)$, is represented context-sensitively [27, 35] by

$$(c', o) \quad \in \quad pt(c, v), \tag{1}$$

indicating that an object o under a heap context c' is pointed to by v under a method context c of m. When m is analyzed under $c = [o_{k-1}, \ldots, o_0]$, $[o_{k-2}, \ldots, o_0]$ will be the heap context for the object created at every allocation site in m.

Figure 1 illustrates *k-obj*, highlighting the combinatorial explosion of contexts in a large program. In Figure 1(a), there is a total of $\frac{r^k-1}{r-1}$ methods, $\texttt{foo}_{i,j}()$, where $0 \leqslant i < k$, $0 \leqslant j < r^{i-1}$ and $r > 1$, in class `A` (which is not given). Note that $\texttt{foo}_{i,j}()$ calls $\texttt{foo}_{i-1,j\div r}()$—that is, $\texttt{foo}_{i,j}()$ has exactly r callers $\texttt{foo}_{i+1,rj}(), \ldots, \texttt{foo}_{i+1,rj+r-1}()$, where $0 < i < k$. In $\texttt{foo}_{i,j}()$, where $0 < i < k$, `a` points to an object, $\texttt{A}_{i,j}$, created at its allocation site `new A()`, which is used as a receiver to call $\texttt{foo}_{i-1,j\div r}()$. In $\texttt{foo}_{0,0}()$, `a` points to an object, $\texttt{A}_{0,0}$, created at its `new A()`, which is passed to as an

argument to `goo()` in the call `b.goo(a)`. In `goo()`, `bar()` is called on the receiver $\mathsf{A}_{0,0}$ pointed by its formal parameter `p`.

Figure 1(b) depicts graphically a total of r^{k-1} method contexts (counted as the number of leaves in the inverted tree), which form the set C_{bar} defined in the following, used for analyzing `bar()` invoked at the call `p.bar()`:

$$C_{\text{bar}} \quad = \quad \{[\mathsf{A}_{k-1,j}, \mathsf{A}_{k-2,j\div r}, \ldots, \mathsf{A}_{1,j\div r^{k-2}}, \mathsf{A}_{0,0}] \mid 0 \leqslant j < r^{k-1}\}, \tag{2}$$

where $[\mathsf{A}_{k-1,j}, \mathsf{A}_{k-2,j\div r}, \ldots, \mathsf{A}_{1,j\div r^{k-2}}]$ is a heap context of $\mathsf{A}_{0,j\div r^{k-1}}$, which happens to be $\mathsf{A}_{0,0}$. Due to how a heap context is related with its corresponding method context on *k-obj*, fixing one will also fix the other.

2.2 Eagle: Partial Context Sensitivity

Before introducing our Eagle approach, we wish to emphasize that the precision of a context-sensitive pointer analysis, once completed, is often measured with all contexts dropped as follows:

$$\overline{pt}(v) \quad = \quad \cup_{c \,\in\, mtx(v)}\{o \mid (c', o) \in pt(c, v)\}, \tag{3}$$

where $\overline{pt}(v)$ is the context-insensitive points-to set of v obtained by merging all context-sensitive points-to sets $pt(c, v)$ given in (1), with c ranging over $mtx(v)$—that is, the set of all method contexts c of v. In this article, as in the literature [11, 12, 17, 36, 39, 40], different context-sensitive pointer analyses will be compared in terms of end-user visible (i.e., context-insensitive) metrics for a number of well-known clients, such as call-graph construction, may-fail-casting, and polymorphic call detection, based on the end-user context-insensitive points-to information thus obtained.

For a given context-sensitive pointer analysis P, we sometimes distinguish the context-sensitive and context-insensitive points-to sets of a variable v obtained by P by writing $pt_P(c, v)$ and $\overline{pt}_P(v)$, respectively. In both cases, we will drop the subscript P whenever the context is clear about P being discussed.

We first describe our insights for empowering *k-obj* with partial context sensitivity (Section 2.2.1), then our pre-analysis for enforcing it (Section 2.2.2), and finally some challenges faced in developing our approach (Section 2.2.3).

2.2.1 Insights. Eagle first applies a pre-analysis to a program once and then leverages it to speed up its subsequent main analysis, *k-obj*, context-sensitively only for parts of the program selected (identically for all values of k). We focus on how Eagle analyzes `bar()` given in Figure 1, which contains z as a variable and X as an allocation site.

As is well known, *k-obj* analyzes every method context-sensitively with k-element contexts uniformly across the program. Thus, `bar()` in Figure 1(a) will be analyzed under r^{k-1} contexts in C_{bar} given in (2), as shown in Figure 1(b). As such, *k-obj* does not scale beyond $k = 2$ for reasonably large programs [11, 36, 40].

Eagle is designed to accelerate *k-obj* without any loss of precision for the first time. Eagle will enable *k-obj* to analyze a method with partial context sensitivity by making *k-obj* run significantly faster while maintaining its precision (by guaranteeing that $\overline{pt}_{\text{Eagle}}(v) = \overline{pt}_{k\text{-}obj}(v)$ for every variable v). Eagle achieves this by performing a pre-analysis, based on a new CFL-reachability formulation of *k-obj*, to identify which variables/allocation sites in a method should be handled by *k-obj* context-sensitively or context-insensitively. Consider `bar()` in Figure 1 again. For its variable z, *k-obj* will be asked to compute r^{k-1} points-to sets in $\{pt([A_{k-1,j_{k-1}}, \ldots, A_{1,j_1}, \mathsf{A}_{0,0}], z) \mid [A_{k-1,j_{k-1}}, \ldots, A_{1,j_1}, \mathsf{A}_{0,0}] \in C_{\text{bar}}\}$ if z is selected to be context-sensitive, but only one point-to set, $pt([\,], z)$, otherwise. For its allocation site X, *k-obj* will be asked to create r^{k-2} abstractions of X in

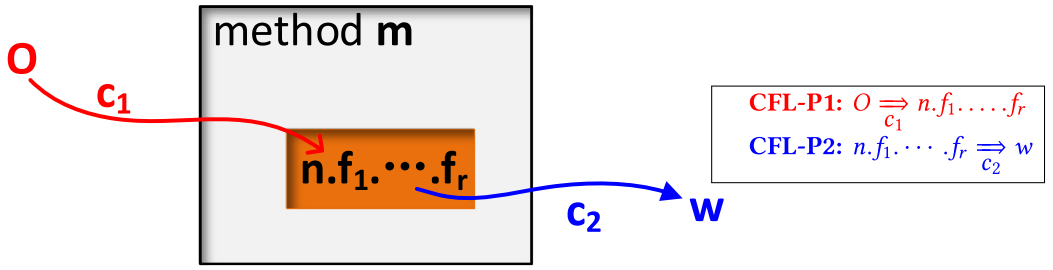


Fig. 2. A CFL-reachability-based condition used in Eagle's pre-analysis for selecting a variable/allocation site n in a method m to be analyzed by k-*obj* context-sensitively if CFL − P1 ∧ CFL − P2 holds. $O \underset{c_1}{\Rightarrow} n.f_1.....f_r$ and $n.f_1.....f_r \underset{c_2}{\Rightarrow} w$ (depicted as the arrows on the left and right of m, respectively) are the inter-procedural value flows defined in Section 2.2.2, where O represents an object, w a variable, $f_1.....f_r$ a sequence of field names, and c_1 and c_2 two nonempty contexts of m (which may be identical).

$\{[A_{k-2,j_{k-2}}, \ldots, A_{1,j_1}, \mathsf{A}_{0,0}, X] \mid [A_{k-1,j_{k-1}}, \ldots, A_{1,j_1}, \mathsf{A}_{0,0}] \in C_{\text{bar}}\}$ if X needs to be modeled context-sensitively, but only one abstraction, X, otherwise.

2.2.2 CFL-Reachability-Based Pre-Analysis. We motivate our CFL-reachability-based pre-analysis by showing how it works through reasoning about the value flow in two small example programs. Our pre-analysis aims to inform k-*obj* about whether it should analyze a generic variable/allocation site n in a method m context-sensitively or not by verifying a CFL-reachability-based condition that governs the value flow into and out of n. To enable k-*obj* to preserve precision under such partial context sensitivity, our pre-analysis will be conservative by recognizing all possible value flows into and out of n in the program, and possibly more spuriously (to err on the side of analyzing n context-sensitively). In Section 3, we will formalize it in terms of a new CFL-reachability formulation of k-*obj* on a **Pointer Assignment Graph (PAG)**, G_{pag}, with its nodes representing the variables/fields/allocation sites in the program.

As k-*obj* is context-sensitive but flow-insensitive, our pre-analysis reasons about the value flow in a program by considering only its data dependences, context-sensitively (by distinguishing method (calling) contexts with their receiver objects) and flow-insensitively (by ignoring the flow of control). As is standard, all variables in a method are assumed to be in SSA (Static Single Assignment) form. By convention, a variable/allocation site (i.e., object) n in a method m is said to be a (*direct* or *indirect*) use of an object O flowing from another method m' (where m and m' may not be necessarily different in the case of recursion) if n affects O stored into an access path [6, 13] $n.f_1.....f_r$ inter-procedurally under some context c of m, denoted $O \underset{c}{\Rightarrow} n.f_1.....f_r$ (i.e., if n appears in the forward slice of O). Similarly, a variable/allocation site n in a method m is said to be a (*direct* or *indirect*) definition of another variable w in another method m'' (where m and m'' may not be necessarily different) if n affects some value stored into w from an access path $n.f_1.....f_r$ inter-procedurally under some context c of m, denoted $n.f_1.....f_r \underset{c}{\Rightarrow} w$ (i.e., if n appears in the backward slice of w). These two predicates will be validated based on the value-flow reachability information computed by our pre-analysis. Due to aliases, all writes into and reads from $n.f_1.....f_r$ can happen outside n's containing method, m.

Figure 2 gives and illustrates our CFL-reachability-based condition for supporting partial context sensitivity: a variable/allocation site n in a method m will be analyzed by k-*obj* context-sensitively if CFL-P1 : $O \underset{c_1}{\Rightarrow} n.f_1.....f_r \wedge$ CFL-P2 : $n.f_1.....f_r \underset{c_2}{\Rightarrow} w$ holds and context-insensitively otherwise. Note that c_1 and c_2 may or may not be identical, as checking for the

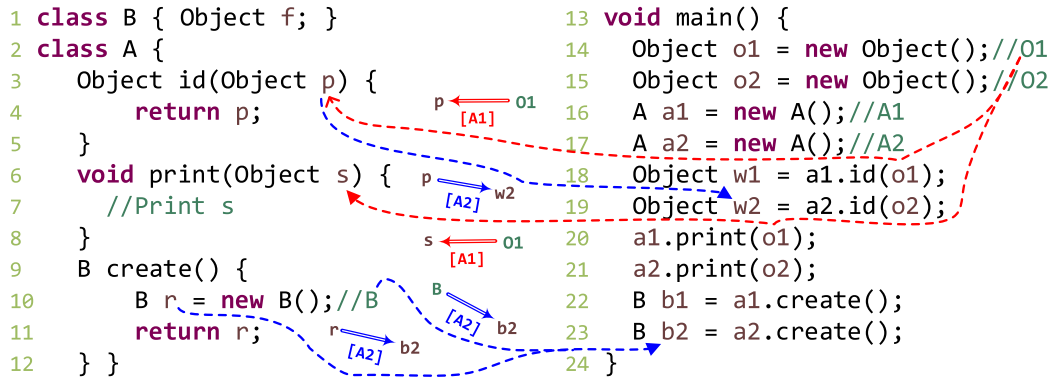


Fig. 3. A program for illustrating the CFL-reachability-based condition given in Figure 2, with the dashed lines depicting how its predicates, CFL – P1 and CFL – P2, are satisfied by the variables/allocation sites in the three methods in class A.

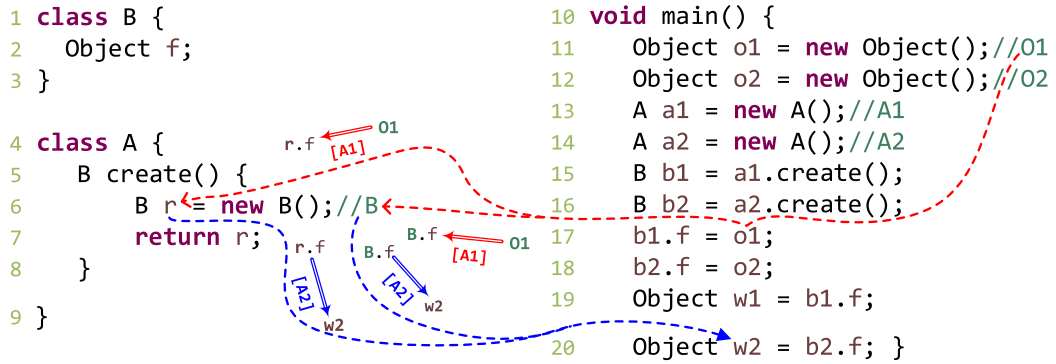


Fig. 4. A modified version of the program given in Figure 3 for illustrating the CFL-reachability-based condition, CFL – P1 and CFL – P2, given in Figure 2, showing that the variable r and the allocation site B must now be analyzed context-sensitively for *2-obj* to maintain its precision for the two points-to sets $\overline{pt}_{2\text{-}obj}(\texttt{w1})$ and $\overline{pt}_{2\text{-}obj}(\texttt{w2})$.

existence of at least two different inter-procedural value flows across a method under two different contexts is unnecessary, in practice. To preserve precision, our pre-analysis can over-approximate but not under-approximate the value flow in the program. In the following, we illustrate this condition with two small programs given in Figures 3 and 4, where the latter example is a slight modification of the former example.

Let us consider the program in Figure 3 first. Table 1 gives the points-to results computed by the algorithm of Andersen [1] context-insensitively and *1-obj* context-sensitively. Let $\overline{pt}_{\text{Andersen}}(v)$ be the points-to set of v computed by Andersen's analysis. As highlighted (in bold), *1-obj* is more precise as the context-insensitive points-to sets of w1 and w2 computed by *1-obj* are more precise: $\overline{pt}_{\text{Andersen}}(\texttt{w1}) = \{\texttt{01}, \texttt{02}\}$ but $\overline{pt}_{1\text{-}obj}(\texttt{w1}) = \{\texttt{01}\}$ and $\overline{pt}_{\text{Andersen}}(\texttt{w2}) = \{\texttt{01}, \texttt{02}\}$ but $\overline{pt}_{1\text{-}obj}(\texttt{w2}) = \{\texttt{02}\}$. Based on our pre-analysis as discussed in the following, *1-obj* will still produce the same points-to results if its Eagle-guided version (denoted *ek-obj*) analyzes only p in id() context-sensitively (and all remaining variables/allocation sites context-insensitively) so that $\overline{pt}_{e1\text{-}obj}(\texttt{w1}) = \overline{pt}_{1\text{-}obj}(\texttt{w1}) = \{\texttt{01}\}$ and $\overline{pt}_{e1\text{-}obj}(\texttt{w2}) = \overline{pt}_{1\text{-}obj}(\texttt{w2}) = \{\texttt{02}\}$.

Table 1. Points-to Results Computed by Andersen's Analysis and *1-obj* for the Program in Figure 3

Method	Andersen's Analysis (Points-to Sets)	*1-obj* (Points-to Sets)	
		Context-Sensitive	Context-Insensitive
main()	o1 : {o1} o2 : {o2} a1 : {A1} a2 : {A2} w1 : {o1, **o2**} w2 : {**o1**, o2} b1 : {B} b2 : {B}	([ ], o1) : {o1} ([ ], o2) : {o2} ([ ], a1) : {A1} ([ ], a2) : {A2} ([ ], w1) : {o1} ([ ], w2) : {o2} ([ ], b1) : {B} ([ ], b2) : {B}	o1 : {o1} o2 : {o2} a1 : {A1} a2 : {A2} w1 : {o1} w2 : {o2} b1 : {B} b2 : {B}
id()	p : {o1, o2}	([A1], p) : {o1} ([A2], p) : {o2}	p : {o1, o2}
print()	s : {o1, o2}	([A1], s) : {o1} ([A2], s) : {o2}	s : {o1, o2}
create()	r : {B}	([A1], r) : {B} ([A2], r) : {B}	r : {B}

Note that $pt(c, v) = S$ and $\overline{pt}(v) = S$ are abbreviated to $(c, v) : S$ and $v : S$, respectively. The two objects, **O1** and **O2**, highlighted in bold are spurious.

How do we make these context-selectivity-related choices systematically? Class A contains three instance methods: `id()` is an identity function that returns whatever its parameter p receives, `print()` simply prints the parameter received (but does not return anything to its callers), and `create()` returns a newly allocated instance of B whenever it is called (but does not receive anything from its callers). Given this program, different pointer analyses differ in terms of the amount of context sensitivity applied in the program and the precision obtained for the points-to sets of w1 and w2.

Context-insensitively, Andersen's analysis finds the points-to sets in this example program given in Table 1 (Column 2), with one points-to set per variable. Without distinguishing the two contexts under which `id()` is called (in lines 18 and 19), the objects pointed by p end up being merged: $\overline{pt}_{\text{Andersen}}(\text{p}) = \{\text{o1}, \text{o2}\}$. Therefore, $\overline{pt}_{\text{Andersen}}(\text{w1}) = \overline{pt}_{\text{Andersen}}(\text{w2}) = \{\text{o1}, \text{o2}\}$, causing w1 to point to o2 and w2 to point to o1 spuriously.

Context-sensitively, *1-obj* distinguishes the calling contexts of a method by its receivers, yielding the points-to sets in Table 1 (Columns 3 and 4). By analyzing `id()` under [A1] (for the call in line 18) and under [A2] (for the call in line 19), *1-obj* obtains $pt_{1\text{-}obj}([\text{A1}], \text{p}) = \{\text{o1}\}$ and $pt_{1\text{-}obj}([\text{A2}], \text{p}) = \{\text{o2}\}$, and consequently, $pt_{1\text{-}obj}([\,], \text{w1}) = \{\text{o1}\}$ and $pt_{1\text{-}obj}([\,], \text{w2}) = \{\text{o2}\}$. Now, $\overline{pt}_{1\text{-}obj}(\text{w1}) = \{\text{o1}\}$ and $\overline{pt}_{1\text{-}obj}(\text{w2}) = \{\text{o2}\}$ without the spurious points-to information introduced by Andersen's analysis. However, *1-obj* has analyzed also `print()` (`create()`) context-sensitively for its two callsites in lines 20 and 21 (22 and 23) unnecessarily without achieving any precision gain, as revealed in Table 1.

We can now apply our pre-analysis to determine which variables/allocation sites in the program given in Figure 3 should be analyzed by *1-obj* context-sensitively or context-insensitively while still maintaining its precision, by checking our CFL-reachability condition CFL-P1 ∧ CFL-P2 as illustrated by the arrowed lines. We only need to consider the three instance methods in class A as the variables/allocation sites in `main()` do not require context sensitivity. As illustrated, p in `id()` should be context-sensitive, as both CFL-P1 : $\text{o1} \xRightarrow[\text{[A1]}]{} \text{p}$ and CFL-P2 : $\text{p} \xRightarrow[\text{[A2]}]{} \text{w2}$ hold. The information that flows into p context-sensitively from outside id() also flows out of id() context-sensitively. As discussed earlier, applying context sensitivity to p is essential to separate

Table 2. Points-to Results Computed by Andersen's Analysis and *2-obj* for the Program in Figure 4

Method	Andersen's Analysis (Points-to Sets)	*2-obj* (Points-to Sets)	
		Context-Sensitive	Context-Insensitive
main()	o1 : {O1} o2 : {O2} a1 : {A1} a2 : {A2} w1 : {O1, **O2**} w2 : {**O1**, O2} b1 : {B} b2 : {B}	([ ], o1) : {O1} ([ ], o2) : {O2} ([ ], a1) : {A1} ([ ], a2) : {A2} ([ ], w1) : {O1} ([ ], w2) : {O2} ([ ], b1) : {[A1], B} ([ ], b2) : {[A2], B}	o1 : {O1} o2 : {O2} a1 : {A1} a2 : {A2} w1 : {O1} w2 : {O2} b1 : {B} b2 : {B}
create()	r : {B}	([A1], r) : {[A1], B} ([A2], r) : {[A2], B}	r : {B}

Note that $pt(c, v) = S$ and $\overline{pt}(v) = S$ are abbreviated to $(c, v) : S$ and $v : S$, respectively. The two objects highlighted in bold are spurious.

its context-sensitive value flows, and consequently maintain the precision for $\overline{pt}_{1\text{-}obj}$(w1) and $\overline{pt}_{1\text{-}obj}$(w2). As for s in print(), context sensitivity is not needed, since CFL-P1 : O1 $\xRightarrow[\text{[A1]}]{}$ s holds but CFL-P2 does not. The information that flows into s from outside print() context-sensitively never flows out of print() at all. As for the variable r and allocation site B in create(), the situation is reversed. Their handling can also be context-insensitive, since CFL-P2 is satisfied (r $\xRightarrow[\text{[A2]}]{}$ b2 for r and B $\xRightarrow[\text{[A2]}]{}$ b2 for B) but CFL-P1 is not. In this case, there is information flowing out of r and B context-sensitively but not any flowing into r and B from outside create(). Thus, by analyzing only p context-sensitively, Eagle guarantees that *e1-obj* achieves exactly the same precision as *1-obj*, and consequently $\overline{pt}_{e1\text{-}obj}$(w1) = $\overline{pt}_{1\text{-}obj}$(w1) = {O1} and $\overline{pt}_{e1\text{-}obj}$(w2) = $\overline{pt}_{1\text{-}obj}$(w2) = {O2}.

Whether a variable/allocation site is context-sensitive or not must be determined by how it is accessed anywhere in the program (in the presence of aliases). If we modify main() in Figure 3 as shown in Figure 4, r and B must now be analyzed context-sensitively. There is some information flowing into and out of r.f and B.f context-sensitively (even though all accesses to B.f happen outside create()): CFL-P1 : O1 $\xRightarrow[\text{[A1]}]{}$ r.f $\wedge$ CFL-P2 : r.f $\xRightarrow[\text{[A2]}]{}$ w2 for r and CFL-P1 : O1 $\xRightarrow[\text{[A1]}]{}$ B.f $\wedge$ CFL-P2 : B.f $\xRightarrow[\text{[A2]}]{}$ w2 for B. To compute the points-to sets of w1 and w2 precisely, *2-obj* is needed now. Table 2 gives the points-to results computed by the algorithm of Andersen [1] context-insensitively and 2-*obj* context-sensitively. Context-sensitively, $pt_{2\text{-}obj}$([A1], r) = {([A1], *B*)} and $pt_{2\text{-}obj}$([A2], r) = {([A2], *B*)}. As a result, $pt_{2\text{-}obj}$([ ], b1) = {([A1], *B*)} and $pt_{2\text{-}obj}$([ ], b2) = {([A2], *B*)}, implying that $\overline{pt}_{2\text{-}obj}$(w1) = {O1} and $\overline{pt}_{2\text{-}obj}$(w2) = {O2}. However, if r and B are analyzed context-insensitively instead, then $pt_{2\text{-}obj}$([ ], r) = {B}. As a result, $pt_{2\text{-}obj}$([ ], b1) = $pt_{2\text{-}obj}$([ ], b2) = {B}, giving rise to less precise points-to sets: $\overline{pt}_{2\text{-}obj}$(w1) = $\overline{pt}_{2\text{-}obj}$(w2) = {O1, O2}. Thus, by analyzing only r and B context-sensitively, Eagle guarantees that *e2-obj* achieves exactly the same precision as *2-obj*, and consequently $\overline{pt}_{e2\text{-}obj}$(w1) = $\overline{pt}_{2\text{-}obj}$(w1) = {O1} and $\overline{pt}_{e2\text{-}obj}$(w2) = $\overline{pt}_{2\text{-}obj}$(w2) = {O2}. Note that Eagle guarantees that *ek-obj* has exactly the same precision as *k-obj* (for any value of *k*).

2.2.3 Technical Challenges. There are three challenges faced in developing our Eagle-style pre-analysis:

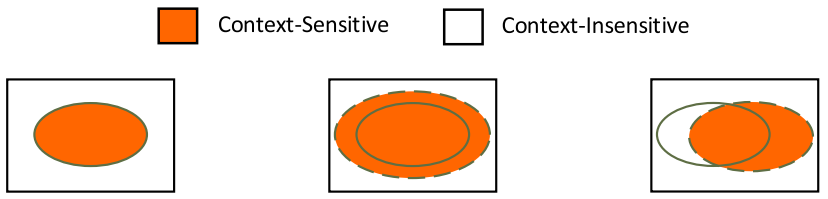


Fig. 5. Three types of pre-analyses for supporting partial context sensitivity in analyzing a method (represented by a box). In (a), the oracle pre-analysis identifies exactly the set of variables/allocations (depicted as a shaded oval) in the method that must be analyzed by *k-obj* context-sensitively for *k-obj* not to lose precision. In (b), the pre-analysis is precision-preserving as some variables/allocation sites that can be analyzed context-insensitively are analyzed context-sensitively. In (c), the pre-analysis is not precision-preserving as some variables/allocation sites that should be analyzed context-sensitively are analyzed context-insensitively.

- First, our pre-analysis, which determines if *k-obj* should analyze a variable/allocation site context-sensitively or not, needs to reason about the value flow into and out of this variable/allocation site *object-sensitively* across the program, as illustrated in Figure 2. This has never been done before. We will tackle this first challenge by introducing a new CFL-reachability formulation of *k-obj* so that we can conduct the required value-flow analysis in terms of CFL-reachability. Our CFL-reachability formulation for object sensitivity is fundamentally different from the CFL-reachability formulation that applies only to callsite sensitivity [5, 33, 37, 41, 45].
- Second, our pre-analysis aims to enable *k-obj* to analyze certain variables/allocation sites in the program context-insensitively so as to reduce the number of context-sensitive facts inferred by *k-obj* (by being effective) while still ensuring that *k-obj* computes the same points-to information as before (by being precision-preserving), as illustrated in Figure 5. These are two conflicting requirements. On one hand, a pre-analysis that asks *k-obj* to analyze everything context-sensitively (with the subsequent main analysis being *k-obj* itself) will certainly be precision-preserving but ineffective. On the other hand, a pre-analysis that asks *k-obj* to analyze everything context-insensitively (with the subsequent main analysis being Andersen's analysis [1]) will certainly be effective but non-precision-preserving. Ideally, as shown in Figure 5(a), our pre-analysis should identify exactly the set $\mathcal{S}$ of variables/allocation sites in a method *m* that must be analyzed by *k-obj* context sensitivity without any precision loss. In practice, our pre-analysis preserves precision by requesting *k-obj* to analyze a non-strict superset of $\mathcal{S}$ in *m* context-sensitively as in Figure 5(b). In contrast, the pre-analysis in Figure 5(c) does not preserve precision as it asks *k-obj* to analyze a strict subset of $\mathcal{S}$ in *m* context-sensitively. We will address this second challenge by developing an effective precision-preserving pre-analysis for solving a CFL-reachability problem fully context-sensitively (without *k*-limiting), based on our new CFL-reachability formulation of *k-obj*.
- Finally, any pre-analysis derived directly from our CFL-reachability formulation of *k-obj* will be undecidable [32] without *k*-limiting and polynomially solvable even with *k*-limiting [14]. We will address this third challenge by regularizing one CFL for specifying field accesses and maintaining another CFL for specifying context-sensitive method calls (without *k*-limiting) so that our pre-analysis will remain fully context-sensitive but become linearly solvable in terms of the number of pointer assignment edges in the program.

$$
\begin{array}{rcll}
\text{Class} & ::= & \text{class } C \text{ extends } C \{ \overline{C\ f}\ ; \overline{M} \} & \\
M \in \text{Method} & ::= & C\ m\ (\ \overline{C\ v}\ ) \{ \overline{C\ v}; \overline{S}; \text{return } v; \} & \\
S \in \text{Statement} & ::= & v = \text{new } C(\overline{v})\ ;^{\ell} & [\textsc{New}] \\
& | & v = v\ ; & [\textsc{Assign}] \\
& | & v.f = v\ ; & [\textsc{Store}] \\
& | & v = v.f\ ; & [\textsc{Load}] \\
& | & v = v.m(\overline{v})\ ; & [\textsc{Invoke}]
\end{array}
$$

$v \in$ Var is a set of variable names
$C \in$ ClassName is a set of class names
$m \in$ MethodName is a set of method names
$f \in$ FieldName is a set of field names
$\ell \in$ Alloc is a set of labels for allocation sites

Fig. 6. A simplified Java language.

3 EAGLE: CFL-REACHABILITY-GUIDED PARTIAL CONTEXT SENSITIVITY

We formalize our precision-preserving pre-analysis for enabling *k-obj* with partial context sensitivity based on CFL-reachability. Section 3.1 gives a simple object-oriented language for formalizing our analysis. Section 3.2 discusses how to represent the value flow in a program with an object-sensitive PAG. Section 3.3 gives a new CFL-reachability formulation of *k-obj* that, once *k*-limited, achieves the same precision as *k-obj*. Section 3.4 describes our lightweight precision-preserving pre-analysis, based on this CFL-reachability formulation of *k-obj*.

3.1 A Simplified Java Language

Figure 6 gives a simple object-oriented language (i.e., a small subset of Java) that contains all operations analyzed by *k-obj*. All fields are instance fields, and all methods are instance methods. Methods are stylized to always have a single return statement. Constructors are regarded as regular instance methods. Section 4 describes how static fields and methods are handled. An allocation site is identified by its label (e.g., line number). Therefore, primitive types are ignored. As *k-obj* is context-sensitive but flow-insensitive, all control flow statements are also elided.

In our formalization, the parameters of an (instance) method m are uniquely identified as follows. The "this" pointer and i-th parameter of m are denoted by $this^m$ and p_i^m, respectively. In addition, a pseudo local variable ret^m in m is assumed to store the return value of m (if any). In our CFL-reachability formulation of *k-obj*, ret^m actually serves as a special parameter of m with the value flow not only leaving ret^m to outside but also entering ret^m from outside.

In our small language, there are only five different types of statements, New, Assign, Store, Load, and Invoke, analyzed by *k-obj*. To support object-sensitive pointer analysis, passing parameters and return values will be modeled as Store and Load statements, respectively, for the first time as discussed in the following.

3.2 Object-Sensitive PAGs

All context-sensitive pointer analyses that were previously formulated in terms of CFL-reachability [5, 33, 37, 41, 45] operate on a graph representation of the program—PAG. However, only callsite sensitivity is assumed. It is achieved over a balanced-parentheses language by matching call and return edges, labeled by their callsites' line numbers (which can be statically determined), with passing parameters and return values treated as Assign statements. However, this treatment for callsite sensitivity does not carry over to object sensitivity.

Statement	Edges	Kind
$x = \text{new } C() \; //O$	$O \xrightarrow{\text{new}} x$	[New]
$x = y$	$y \xrightarrow{\text{assign}} x$	[Assign]
$x.f = y$	$y \xrightarrow{\text{store}[f]} x$; $\dfrac{O \in \overline{pt}(x)}{O \xrightarrow{\text{hload}[f]} f \;\; \widehat{O}}$	[Store]
$x = y.f$	$y \xrightarrow{\text{load}[f]} x$; $\dfrac{O \in \overline{pt}(y)}{f \xrightarrow{\text{hstore}[f]} O \;\; \breve{O}}$	[Load]
$x = y.m(..., a_i, ...)$	$\dfrac{O \in \overline{pt}(y),\ m' = dispatch(m, O)}{\cdots}$	[Invoke]

$$y \xrightarrow{\text{store}[this^{m'}]} y \qquad a_i \xrightarrow{\text{store}[p_i^{m'}]} y \qquad y \xrightarrow{\text{load}[ret^{m'}]} x$$

$$O \xrightarrow[\widehat{O}]{\text{hload}[this^{m'}]} this^{m'} \qquad O \xrightarrow[\widehat{O}]{\text{hload}[p_i^{m'}]} p_i^{m'} \qquad ret^{m'} \xrightarrow[\breve{O}]{\text{hstore}[ret^{m'}]} O$$

Fig. 7. Rules for building the object-sensitive PAG for a program (with five kinds of statements). Here, $\overline{pt}(v)$ denotes the points-to set of v pre-computed by a less precise pointer analysis (e.g., Andersen's analysis [1]).

To obtain our pre-analysis as illustrated in Figures 2 and 5, we will first give a new CFL-reachability formulation of *k-obj* under object sensitivity that, once k-limited, has exactly the same precision as *k-obj*, based on a new object-sensitive PAG representation of a program. Therefore, our PAG representation for supporting object sensitivity, as illustrated by the example programs in Figures 3 and 4, is fundamentally different from the one adopted earlier for supporting callsite sensitivity [5, 33, 37, 41, 45]. Unlike callsite sensitivity, which distinguishes the calling contexts of a method by its callsites, object sensitivity distinguishes the calling contexts of a method by its receiver objects. This leads to two observations on how an object-sensitive PAG should be constructed and how *k-obj* should be formulated in terms of CFL-reachability. First, whereas object sensitivity distinguishes the receiver objects in method calls to achieve context sensitivity, field sensitivity must distinguish the base objects of field accesses to achieve field sensitivity, which can be regarded as a form of object-sensitive context sensitivity in a unified manner. Second, passing parameters and returning values for methods can be treated as field stores and loads, respectively, on their receiver objects.

Figure 7 gives an algorithm in terms of a set of rules for building the (object-sensitive) PAG for a program, denoted G_{pag}, whose nodes represent variables, fields, and heap objects, and whose edges represent the value flow through assignments. The rules for adding $O \xrightarrow{\text{new}} x$, $y \xrightarrow{\text{assign}} x$, $y \xrightarrow{\text{store}[f]} x$, and $y \xrightarrow{\text{load}[f]} x$, which are also used for supporting callsite sensitivity, are standard. It suffices to describe only the edges added in the three boxes as part of the three rules, [Store], [Load], and [Invoke], for supporting object sensitivity. In our rules, we can pre-compute $\overline{pt}(v)$

```
Object f(Object f) {
    return f;
}

x.f(y); //x.f = y;
x = y.f(null); //x = y.f;
```

Fig. 8. Object-sensitive modeling of field stores and loads as method calls.

by using the analysis of Andersen [1] and build the PAG in advance. Note that the spurious PAG nodes and edges introduced will obviously not be traversed by a more precise pointer analysis.

Consider a store $x.f = y$ modeled by an edge $y \xrightarrow{\text{store}[f]} x$. In addition to this edge, for every $O \in \overline{pt}(x)$, we also add another edge $O \xrightarrow[\widehat{O}]{\text{hload}[f]} f$—that is, an edge $f = O.f$ labeled with a context $\widehat{O}$ to indicate that the content in $O.f$ is loaded into f under $\widehat{O}$. Consider a load $x = y.f$ modeled by an edge $y \xrightarrow{\text{load}[f]} x$. In addition to this edge, for every $O \in \overline{pt}(y)$, we also add another edge $f \xrightarrow[\breve{O}]{\text{hstore}[f]} O$—that is, $O.f = f$ labeled with a context $\breve{O}$ to indicate that the content in f is stored into $O.f$ under $\breve{O}$. Note that in our PAG representation, fields are treated as a special case of variables uniformly.

Finally, consider a call $x = y.m(\ldots, a_i, \ldots)$. For every receiver $O \in \overline{pt}(y)$, let m' be the virtual method resolved for O. Let $p_i^{m'}$ be the corresponding parameter of a_i. Recall that $ret^{m'}$ stores the value returned by m' (Section 3.1). Then the parameter passing for a_i is simply modeled as a store $y.p_i^{m'} = a_i$, resulting in the edges added by applying [Store]. Similarly, the value returning for x is simply modeled as a load $x = y.ret^{m'}$, resulting in the edges added by applying [Load]. Note that the parameters $this^{m'}$ and $p_i^{m'}$ as well as $ret^{m'}$ are modeled as fields of the receiver O object-sensitively.

For a PAG edge, its label above indicates whether it is an assignment or field access and its label below (if it exists) indicates an *inter-context value flow* (where the label represents a base object of a field access or a receiver object of a method call). An edge without a below-edge label represents an *intra-context value flow*. In our PAG representation, an inter-context value flow can be understood as a context-sensitive value flow in an object-sensitive manner, since field stores and loads can be modeled or understood conceptually as method calls as well, as illustrated in Figure 8, where `null` can be initially marked a special `nullobj` in a flow-insensitive pointer analysis such as *k-obj* and ignored later.

Given an object O, there are two kinds of inter-context edges: $\widehat{O}$ (indicated by a hat) and $\breve{O}$ (indicated by a check). Specifically, $\widehat{O}$ (referred to as an *entry context*) represents a value flow for $f = O.f$ (i.e., from $O.f$ to f) for a field f and $\breve{O}$ (referred to as an *exit context*) represents the opposite value flow for $O.f = f$ (i.e., from f to $O.f$). In Section 3.3, we will see how to achieve object-sensitive context sensitivity by matching two kinds of inter-context edges.

Let us apply the rules given in Figure 7 to build the PAGs for the two motivating examples discussed in Section 2:

Figure 3: Figure 9 depicts the (object-sensitive) PAG for the program in Figure 3. For illustration purposes, we assume that this PAG is created based on the points-to information computed by Andersen's analysis given in Table 1. The PAG is symmetric horizontally across `p` as `id()`, `print()`, and `create()` are all invoked on two receivers, `A1` and `A2`. Let us explain how the part of the PAG is constructed for `w1 = a1.id(o1)` in line 18

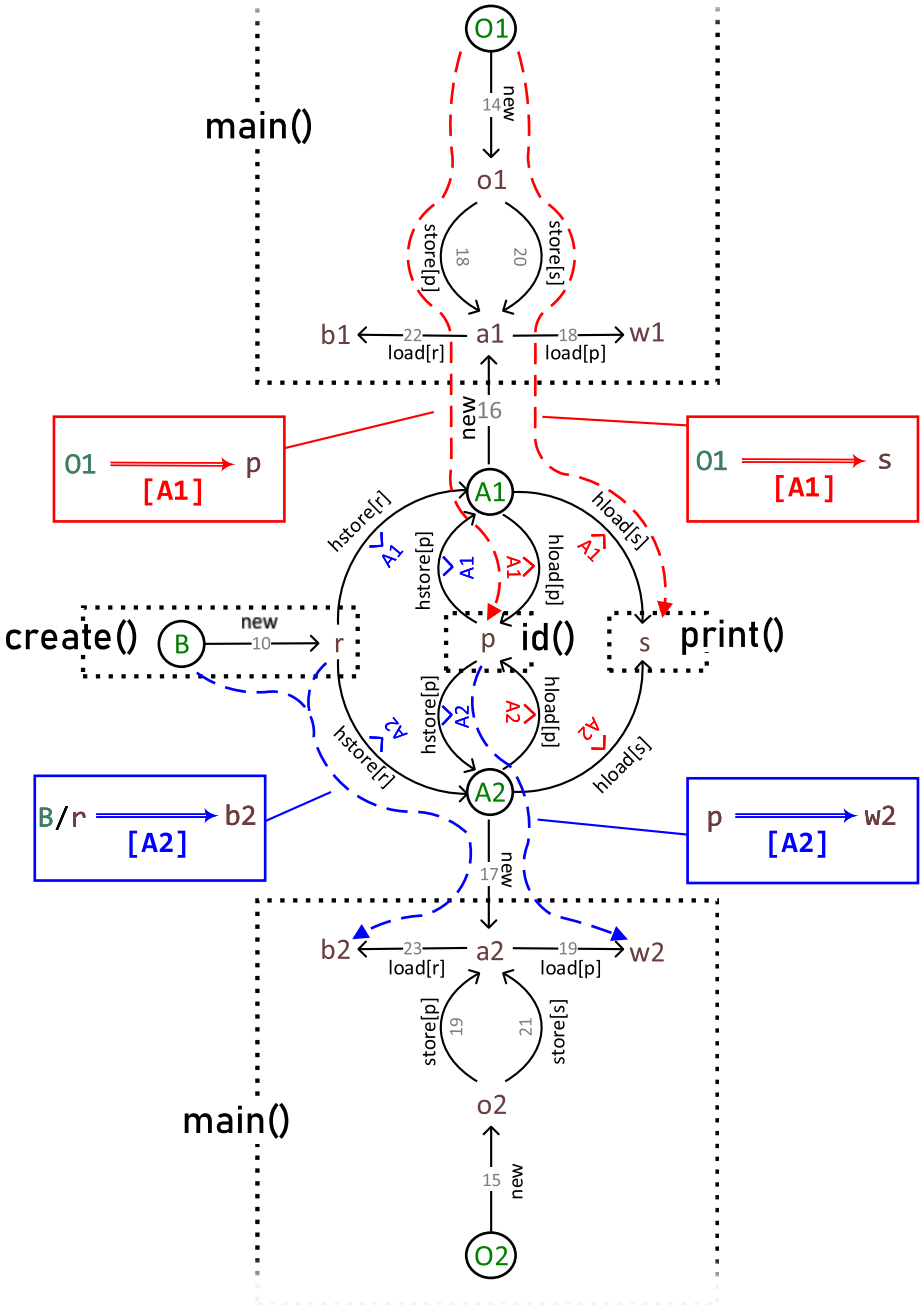


Fig. 9. The PAG for the program given in Figure 3 with the CFL-reachability-based condition made explicit.

invoked under receiver A1 (since A1 $\in \overline{pt}_{\text{Andersen}}$(a1)). In our PAG representation, passing a parameter is modeled as a store, assigning o1 to p is handled as a1.p = o1. According to [Store], we have added o1 $\xrightarrow{\text{store[p]}}$ a1 and A1 $\xrightarrow[\widehat{\text{A1}}]{\text{hload[p]}}$ p since A1 $\in \overline{pt}_{\text{Andersen}}$(a1). This

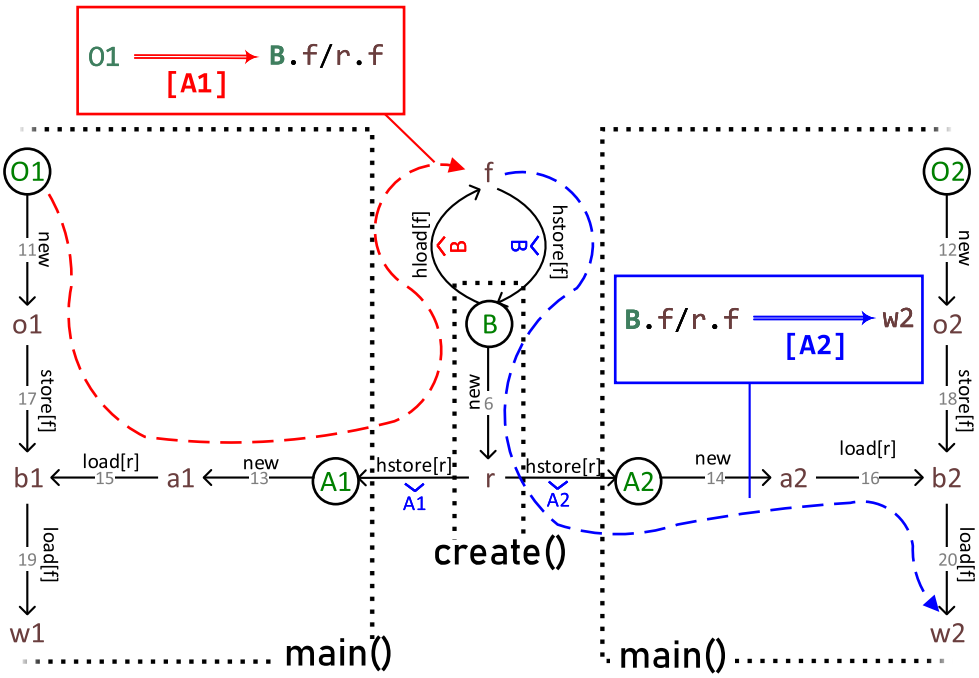


Fig. 10. The PAG for the program in Figure 4 with the CFL-reachability-based condition made explicit.

implies that `p = o1` takes place when `id()` is analyzed with its receiver object being `A1`. As returning a value is modeled as a load, assigning `p` to `w1` is handled as `w1 = a1.p`. According to [Load], we have added $\texttt{w1} \xrightarrow{\text{load[p]}} \texttt{a1}$ and $\texttt{p} \xrightarrow[\widetilde{\texttt{A1}}]{\text{hstore[p]}} \texttt{A1}$ since $\texttt{A1} \in \overline{pt}_{\text{Andersen}}(\texttt{a1})$. This implies that `w1 = p` takes place when `id()` is analyzed with its receiver object being `A1`.

Figure 4: Figure 10 depicts the PAG for the program given in Figure 4, based on the points-to information computed also by Andersen's analysis given in Table 2. This PAG is similar to the part of the PAG related to `create()` given in Figure 9, except we have added the edges for representing the stores in lines 17 and 18 (`b1.f = o1` and `b2.f = o2`) and the loads in lines 19 and 20 (`w1 = b1.f` and `w2 = b2.f`).

In these two PAGs, the (abstract) value flows used for explaining our CFL-reachability-based condition for the original programs in Figures 3 and 4 are now concretized and will be explained later during the development of Eagle.

3.3 A CFL-Reachability Formulation of *k-obj*

We now give a new CFL-reachability formulation of *k-obj* operating on the PAG of a program, G_{pag}. Our new CFL-reachability-based object-sensitive pointer analysis, once k-limited, computes exactly the same points-to information (and thus achieves the same precision) as *k-obj* does. This new form of pointer analysis provides the foundation to enable us to develop a precision-preserving pre-analysis to accelerate *k-obj* with partial context sensitivity.

Let L be a CFL over Σ_{pag}, which is an alphabet drawn from all edge labels in G_{pag}. Each path p in G_{pag} has a label $L(p)$ in Σ^*_{pag} formed by concatenating in order the labels of the edges in p. A node v is *L-reachable* from a node u if there exists a path p from u to v in G_{pag}, called an *L-path*,

such that $L(p) \in L$. To reason about CFL-reachability, we need to traverse the edges in G_{pag} both forward and backward. For each edge $x \xrightarrow[O]{\ell} y$, its inverse edge is $y \xrightarrow[\overline{O}]{\overline{\ell}} x$. To avoid cluttering, the inverse edges in a PAG are not shown explicitly (unless indicated otherwise). For a below-edge label $\widehat{O}$ or $\breve{O}$, we have $\overline{\widehat{O}} = \breve{O}$ and $\overline{\breve{O}} = \widehat{O}$, implying that the concepts of entry and exit contexts (edges) for inter-context value flows are swapped if the original PAG edges are traversed inversely during a particular pointer analysis. For a path p in G_{pag} such that its label is $L(p) = \ell_1, \ldots, \ell_r$ in L, the inverse of p, for instance, $\overline{p}$ has the label $L(\overline{p}) = \overline{\ell_r}, \ldots, \overline{\ell_1}$.

We will define a language, L_{FC}, over Σ_{pag} so that *k-obj* can be solved by reasoning about CFL-reachability in G_{pag}. For convenience, we will express L_{FC} as the intersection of two balanced-parentheses CFLs, $L_{FC} = L_F \cap L_C$, with the two languages specifying two different aspects of *k-obj* as shown in the following. As a result, determining if a variable v points to an object O requires finding a path p between the nodes v and O in G_{pag} such that $L_F(p) \in L_F$ and $L_C(p) \in L_C$:

- $\boldsymbol{L_F}$. L_F ensures that *k-obj* performs a *field-based pointer analysis* context-insensitively. A field-based pointer analysis over-approximates a field-sensitive pointer analysis by ignoring the base objects in all field accesses but still ensuring field accesses are balanced. Given `q = new A() // 0; p.f = q; x = y.f`, `x` is resolved to point to `0` always in a field-based analysis but only when `p` and `y` are aliases in a field-sensitive analysis.
- $\boldsymbol{L_C}$. L_C ensures that *k-obj* is object-sensitive not only for method calls but also for field accesses (by requiring `p` and `y` to be aliases) by matching method calls and returns as a balanced-parentheses problem.

The grammar for L_F given next realizes a context-insensitive field-based pointer analysis for *k-obj*:

$$\begin{array}{rcl}
\mathit{flowsto} & \rightarrow & \mathsf{new}\ \mathit{flows}^* \\
\overline{\mathit{flowsto}} & \rightarrow & \overline{\mathit{flows}}^*\ \overline{\mathsf{new}} \\
\mathit{flows} & \rightarrow & \mathsf{assign} \mid \mathsf{store}[f]\ \overline{\mathit{flowsto}}\ \mathsf{hload}[f] \mid \mathsf{hstore}[f]\ \mathit{flowsto}\ \mathsf{load}[f]\ , \\
\overline{\mathit{flows}} & \rightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{hload}[f]}\ \mathit{flowsto}\ \overline{\mathsf{store}[f]} \mid \overline{\mathsf{load}[f]}\ \overline{\mathit{flowsto}}\ \overline{\mathsf{hstore}[f]}
\end{array} \quad (4)$$

where the set of terminals includes all above-edge labels in G_{pag} (for both regular and their inverse edges in G_{pag}). This means that in defining L_F, all below-edge labels in G_{pag} are ignored.

If O *flowsto* v, then v is L_F-reachable from O (i.e., O flows to v), implying that v points to O. In fact, $\overline{\mathit{flowsto}}$ is the inverse of the *flowsto* relation, representing the standard points-to relation. Thus, if v $\overline{\mathit{flowsto}}$ O, then v points to O. Note that $\overline{\mathit{flows}}$ is also the inverse of *flows*. When computing the points-to information, we need to traverse the edges in G_{pag} forward to establish *flowsto* and *flows* and also backward to establish $\overline{\mathit{flowsto}}$ and $\overline{\mathit{flows}}$, with both performed recursively (due to the need for resolving the aliases in the program).

This is a standard formulation for a program consisting of assignments and field accesses [37], extended significantly for handling the hload and hstore edges to support object sensitivity. By construction, L_F is a balanced-parentheses language since field accesses must be balanced, with `store[f]` matched by `hload[f]`, `hstore[f]` matched by `load[f]`, `hload[f]` matched by `store[f]`, and `load[f]` matched by `hstore[f]`. In the case of *flows*, store$[f]$ $\overline{\mathit{flowsto}}$ hload$[f]$ represents the value flow from a store (say, $p.f = \ldots$) to the field f and hstore$[f]$ *flowsto* load$[f]$ represents the value flow from the *same* field f to a load (say, $\ldots = q.f$). If f is a parameter of a method m', say, $p_i^{m'}$ that is different from $ret^{m'}$, such that there is a value flow from $p_i^{m'}$ to $ret^{m'}$

(which happens if, e.g., in Figure 3, we change "`return p;`" in `id()` (with m' being `id()`) to, say, "`r = p; return r;`", so that $p_i^{m'}$ is `p` and $ret^{m'}$ is `r`), then both $p_i^{m'}$ and $ret^{m'}$ will be connected in the form of $\cdots$ store$[p_i^{m'}]$ $\overline{flowsto}$ hload$[p_i^{m'}]$ $p_i^{m'}$ *flows* $ret^{m'}$ hstore$[ret^{m'}]$ *flowsto* load$[ret^{m'}]\cdots$ according to (4). In this case, $p_i^{m'}$ and $ret^{m'}$ are regarded conceptually as representing a common field of a receiver object of m'. This novel idea of choreographing the value flows around the fields is critical for enabling the object-sensitive handling for both method calls and field accesses in another separate language.

The grammar for L_C given in the following enforces the object-sensitive context sensitivity in *k-obj* for both method calls and field accesses (by therefore turning L_F from a field-based analysis into a field-sensitive analysis):

$$\begin{array}{rcl} realizable & \rightarrow & exit\ entry \\ exit & \rightarrow & exit\ balanced \mid exit\ \breve{O} \mid \epsilon \\ entry & \rightarrow & entry\ balanced \mid entry\ \widehat{O} \mid \epsilon \\ balanced & \rightarrow & balanced\ balanced \mid \widehat{O}\ balanced\ \breve{O} \mid \epsilon \end{array}, \tag{5}$$

where O is an allocation site (as defined in Figure 6). The below-edge labels in G_{pag} form the set of terminals. In defining L_C, all above-edge labels in G_{pag} are ignored. Here, ϵ is the standard symbol denoting the empty string.

This is also a standard formulation for matching method calls and returns by solving a balanced-parentheses problem, except it is introduced for the first time for supporting also object-sensitive pointer analysis for field accesses in a unified manner (Figure 8). A path p in G_{pag} is said to be *realizable* iff it is an L_C-path, with $L_C(p)$ being derived from *realizable* starting with the production *realizable* $\rightarrow$ *exit entry*. In this case, we write $L_C^{exit}(p)$ for the prefix of $L_C(p)$ that is derived by *exit* and $L_C^{entry}(p)$ for the suffix of $L_C(p)$ that is derived by *entry*. An unrealizable path in G_{pag} characterizes a value flow that cannot occur in *k-obj*—for example, a value flow that enters a method (field) on a receiver (base) object but leaves the same method (field) on a different receiver (base) object.

Our CFL-reachability formulation of *k-obj*, defined by $L_{FC} = L_F \cap L_C$, discovers the points-to information in G_{pag} as follows. A path p from O to v in G_{pag} can be identified as a context-sensitive *flowsto* relation iff p is both (1) a *flowsto* path in L_F, where O *flowsto* v, and (2) a realizable path in L_C, where *realizable* $\Longrightarrow^* L_C(p)$, yielding the following context-sensitive points-to relation (using the notation introduced earlier in Section 2):

$$\begin{array}{rcl} (ctx(L_C^{exit}(p)), O) & \in & pt(ctx(L_C^{entry}(p)), v) \\ ctx(L_C^{entry}(p)) & \stackrel{\text{def}}{=\!=} & \text{sequence of unbalanced } \widehat{O'}\text{'sin } L_C^{entry}(p) \text{ with their hats elided} \\ ctx(L_C^{exit}(p)) & \stackrel{\text{def}}{=\!=} & \text{sequence of unbalanced } \breve{O'}\text{'sin } L_C^{exit}(p) \textit{ in reverse} \text{ (i.e., } \overline{L_C^{exit}(p)}) \\ & & \text{with their checks elided} \end{array}, \tag{6}$$

where the method context $ctx(L_C^{entry}(p))$ of v is the sequence of unbalanced entry contexts for the method calls in p and the heap context $ctx(L_C^{exit}(p))$ of O is the sequence of unbalanced exit contexts for the method calls in p in reverse: Note that all entry and exit contexts for the field stores and loads in p in the program are balanced out, as desired.

In the following, we explain how our CFL formulation $L_{FC} = L_F \cap L_C$ works in finding the points-to information in two examples. We use one motivating example first to illustrate L_F and L_C and then a more complex one to illustrate (6).

Let us consider the PAG in Figure 10 constructed for the example in Figure 4. Let us consider L_F first. The *flowsto* relations from `o1` and `o2` to `f`, which correspond to `b1.f = o1` and `b2.f = o2`,

are defined by the following two paths:

$$\begin{aligned} P1 &\overset{\text{def}}{=\!=} \mathsf{o1} \xrightarrow{\mathsf{new}} \mathsf{o1} \xrightarrow{\mathsf{store[f]}} \mathsf{b1} \xrightarrow{\overline{\mathsf{load[r]}}} \mathsf{a1} \xrightarrow{\overline{\mathsf{new}}} \mathsf{A1} \xrightarrow[\widehat{\mathsf{A1}}]{\overline{\mathsf{hstore[r]}}} \mathsf{r} \xrightarrow{\overline{\mathsf{new}}} \mathsf{B} \xrightarrow[\widehat{\mathsf{B}}]{\mathsf{hload[f]}} \mathsf{f} \\ P2 &\overset{\text{def}}{=\!=} \mathsf{o2} \xrightarrow{\mathsf{new}} \mathsf{o2} \xrightarrow{\mathsf{store[f]}} \mathsf{b2} \xrightarrow{\overline{\mathsf{load[r]}}} \mathsf{a2} \xrightarrow{\overline{\mathsf{new}}} \mathsf{A2} \xrightarrow[\widehat{\mathsf{A2}}]{\overline{\mathsf{hstore[r]}}} \mathsf{r} \xrightarrow{\overline{\mathsf{new}}} \mathsf{B} \xrightarrow[\widehat{\mathsf{B}}]{\mathsf{hload[f]}} \mathsf{f}. \end{aligned} \tag{7}$$

These two paths share the same label in L_F: $L_F(P1) = L_F(P2) = \mathsf{new\ store[f]}\ \overline{\mathsf{load[r]}}\ \overline{\mathsf{new}}\overline{\mathsf{hstore[r]}}\ \overline{\mathsf{new}}\ \mathsf{hload[f]}$. The *flowsto* relations from f to w1 and w2 (for w1 = b1.f and w2 = b2.f) are

$$\begin{aligned} Q1 &\overset{\text{def}}{=\!=} \mathsf{f} \xrightarrow[\widecheck{\mathsf{B}}]{\mathsf{hstore[f]}} \mathsf{B} \xrightarrow{\mathsf{new}} \mathsf{r} \xrightarrow[\widecheck{\mathsf{A1}}]{\mathsf{hstore[r]}} \mathsf{A1} \xrightarrow{\mathsf{new}} \mathsf{a1} \xrightarrow{\mathsf{load[r]}} \mathsf{b1} \xrightarrow{\mathsf{load[f]}} \mathsf{w1} \\ Q2 &\overset{\text{def}}{=\!=} \mathsf{f} \xrightarrow[\widecheck{\mathsf{B}}]{\mathsf{hstore[f]}} \mathsf{B} \xrightarrow{\mathsf{new}} \mathsf{r} \xrightarrow[\widecheck{\mathsf{A2}}]{\mathsf{hstore[r]}} \mathsf{A2} \xrightarrow{\mathsf{new}} \mathsf{a2} \xrightarrow{\mathsf{load[r]}} \mathsf{b2} \xrightarrow{\mathsf{load[f]}} \mathsf{w2}. \end{aligned} \tag{8}$$

These two paths also share the same label in L_F: $L_F(Q1) = L_F(Q2) = \mathsf{hstore[f]\ new\ hstore[r]load[r]\ load[f]}$. If we consider just L_F alone for the moment, which performs a context-insensitive field-based pointer analysis (with all below-edge labels ignored), then $P1 + Q1$, $P1 + Q2$, $P2 + Q1$ and $P2 + Q2$ are all *flowsto* paths, since $L_F(P1 + Q1)$, $L_F(P1 + Q2)$, $L_F(P2 + Q1)$, and $L_F(P2 + Q2)$ (merged at f) are all in L_F. This implies that $\overline{pt}(\mathsf{w1}) = \overline{pt}(\mathsf{w2}) = \{\mathsf{o1}, \mathsf{o2}\}$. However, $\mathsf{o2} \in \overline{pt}(\mathsf{w1})$ and $\mathsf{o1} \in \overline{pt}(\mathsf{w2})$ turn out to be spurious (due to the lack of context sensitivity).

However, if we consider also L_C introduced for enforcing object-sensitive context sensitivity, then $P1 + Q1$ and $P2 + Q2$ are realizable but $P1 + Q2$ and $P2 + Q1$ are not. To see this, we find that $L_C(P1 + Q1) = \widehat{\mathsf{A1}}\,\widehat{\mathsf{B}}\,\widecheck{\mathsf{B}}\,\widecheck{\mathsf{A1}}$, $L_C(P1 + Q2) = \widehat{\mathsf{A1}}\,\widehat{\mathsf{B}}\,\widecheck{\mathsf{B}}\,\widecheck{\mathsf{A2}}$, $L_C(P2 + Q1) = \widehat{\mathsf{A2}}\,\widehat{\mathsf{B}}\,\widecheck{\mathsf{B}}\,\widecheck{\mathsf{A1}}$, and $L_C(P2 + Q2) = \widehat{\mathsf{A2}}\,\widehat{\mathsf{B}}\,\widecheck{\mathsf{B}}\,\widecheck{\mathsf{A2}}$. According to L_C, $P1 + Q1$ and $P2 + Q2$ are realizable as they are balanced but $P1 + Q2$ and $P2 + Q1$ are not realizable as $\widehat{\mathsf{A1}}\,\widecheck{\mathsf{A2}}$ and $\widehat{\mathsf{A2}}\,\widecheck{\mathsf{A1}}$ cannot balance out. In Figure 4, o1 can flow to w1 but not w2 and o2 can flow to w2 but not w1 when id() is analyzed context-sensitively.

For the two context-sensitive *flowsto* paths $P1 + Q1$ and $P2 + Q2$, we can use (6) to deduce the underlying points-to information. Given that $ctx(L_C^{entry}(P1 + Q1)) = ctx(L_C^{exit}(P1 + Q1)) = ctx(L_C^{entry}(P2 + Q2)) = ctx(L_C^{exit}(P2 + Q2)) = [\,]$, we obtain $pt([\,], \mathsf{w1}) = \{([\,], \mathsf{o1})\}$ and $pt([\,], \mathsf{w2}) = \{([\,], \mathsf{o2})\}$—that is, $\overline{pt}(\mathsf{w1}) = \{\mathsf{o1}\}$ and $\overline{pt}(\mathsf{w2}) = \{\mathsf{o2}\}$. Thus, the spurious points-to information introduced earlier by considering L_F alone has successfully been eliminated.

Let us consider a more complex example given in Figure 11, with its PAG depicted in Figure 12. This example is designed to help understand how the context-sensitive points-to information for a variable (i.e., v here) is obtained in computing L_{FC} according to (6). By visual inspection, we can see that ([B1, B2], v) points to ([A1, A2], 0).

In the program given in Figure 11, a1a2 points to an object A2 allocated under context [A1], b1b2 points to an object B2 allocated under context [B1], and a1a2o points to an object 0 allocated under context [A1, A2]. Finally, 0 is assigned to the parameter v of receiveObj() invoked under [B1, B2] in propagate(), which is invoked under [C] from main(). To build the PAG shown in Figure 12 according to the rules in Figure 7, we make use of the points-to information pre-computed by the analysis of Andersen [1] (elided here). Its new edges are introduced for the allocation sites by applying [New], and its store, load, hload, and hstore edges are added for the method calls by applying [Invoke].

```
1  class A {
2      A createA() {
3          A a2 = new A(); // A2
4          return a2;
5      }
6      Object createObj() {
7          Object o = new Object(); // O
8          return o;
9      }
10 }
11 class B {
12     B createB() {
13         B b2 = new B(); // B2
14         return b2;
15     }
16     void receiveObj(Object v) { }
17 }
```

```
19 class C {
20     void propagate(Object p, B b) {
21         b.receiveObj(p);
22     }
23 }
24 void main() {
25     A a1 = new A(); // A1
26     A a1a2 = a1.createA();
27     Object a1a2o = a1a2.createObj();
28     B b1 = new B(); // B1
29     B b1b2 = b1.createB();
30     C c = new C(); // C
31     c.propagate(a1a2o,b1b2);
32 }
```

Fig. 11. An example for illustrating the CFL-reachability formulation $L_{FC} = L_F \cap L_C$.

Consider the following path from 0 to v, denoted P, highlighted as a dashed line in the PAG of this example (Figure 12):

$$\begin{aligned} P_{0,\mathsf{v}} \overset{\text{def}}{=\!=} \; & \mathsf{0} \xrightarrow{\text{new}} \mathsf{o} \xrightarrow[\widecheck{\mathsf{A2}}]{\text{hstore[o]}} \mathsf{A2} \xrightarrow{\text{new}} \mathsf{a2} \xrightarrow[\widecheck{\mathsf{A1}}]{\text{hstore[a2]}} \mathsf{A1} \xrightarrow{\text{new}} \mathsf{a1} \xrightarrow{\text{load[a2]}} \mathsf{a1a2} \\ & \xrightarrow{\text{load[o]}} \mathsf{a1a2o} \xrightarrow{\text{store[p]}} \mathsf{c} \xrightarrow{\overline{\text{new}}} \mathsf{C} \xrightarrow[\widehat{\mathsf{C}}]{\text{hload[p]}} \mathsf{p} \xrightarrow{\text{store[v]}} \mathsf{b} \xrightarrow[\widecheck{\mathsf{C}}]{\overline{\text{hload[b]}}} \mathsf{C} \xrightarrow{\text{new}} \mathsf{c}. \\ & \xrightarrow{\overline{\text{store[b]}}} \mathsf{b1b2} \xrightarrow{\overline{\text{load[b2]}}} \mathsf{b1} \xrightarrow{\overline{\text{new}}} \mathsf{B1} \xrightarrow[\widehat{\mathsf{B1}}]{\overline{\text{hstore[b2]}}} \mathsf{b2} \xrightarrow{\overline{\text{new}}} \mathsf{B2} \xrightarrow[\widehat{\mathsf{B2}}]{\text{hload[v]}} \mathsf{v} \end{aligned} \tag{9}$$

The string formed by all labels drawn from L_F can be read off from $P_{0,\mathsf{v}}$ as

$$\begin{aligned} L_F(P_{0,\mathsf{v}}) \; = \; & \texttt{new hstore[o] new hstore[a2] new load[a2] load[o] store[p] } \overline{\texttt{new}} \\ & \texttt{hload[p] store[v] } \overline{\texttt{hload[b]}} \texttt{ new } \overline{\texttt{store[b]}}\ \overline{\texttt{load[b2]}}\ \overline{\texttt{new}}\ \overline{\texttt{hstore[b2]}}. \\ & \overline{\texttt{new}}\ \texttt{hload[v]} \end{aligned} \tag{10}$$

Thus, $P_{0,\mathsf{v}}$ is a *flowsto* path in L_F. In addition, the string formed by the labels drawn L_C can also be read off as

$$L_C(P_{0,\mathsf{v}}) = \widecheck{\mathsf{A2}}\ \widecheck{\mathsf{A1}}\ \widehat{\mathsf{C}}\ \widecheck{\mathsf{C}}\ \widehat{\mathsf{B1}}\ \widehat{\mathsf{B2}}. \tag{11}$$

Thus, $P_{0,\mathsf{v}}$ is also a *realizable* path in L_C. According to (6), we can extract $L_C^{exit}(P_{0,\mathsf{v}}) = \widecheck{\mathsf{A2}}\ \widecheck{\mathsf{A1}}\ \widehat{\mathsf{C}}\ \widecheck{\mathsf{C}}$ and $L_C^{entry}(P_{0,\mathsf{v}}) = \widehat{\mathsf{C}}\ \widecheck{\mathsf{C}}\ \widehat{\mathsf{B1}}\ \widehat{\mathsf{B2}}$ from $L_C(P_{0,\mathsf{v}})$, and subsequently obtain the following points-to relation for v from $P_{0,\mathsf{v}}$ (by noting that $ctx(L_C^{exit}(P_{0,\mathsf{v}})) = [\mathsf{A1},\mathsf{A2}]$ and $ctx(L_C^{entry}(P_{0,\mathsf{v}})) = [\mathsf{B1},\mathsf{B2}]$):

$$([\mathsf{A1},\mathsf{A2}],\mathsf{O}) \in pt([\mathsf{B1},\mathsf{B2}],\mathsf{v}). \tag{12}$$

The following theorem states that performing *k-obj* in terms of CFL-reachability yields the same points-to information.

Theorem 3.1 [Correctness]. *Let* CFL-obj *be the CFL-reachability-based pointer analysis for solving* $L_{FC} = L_F \cap L_C$. *For a program,* CFL-obj *computes exactly the same points-to information under k-limiting as k-*obj, *by operating on* G_{pag}, *which is pre-computed by a less precise analysis (i.e., with* $\overline{pt}(v)$ *in Figure 7 pre-computed over-approximately).*

Proof. L_F enforces the part of *k-obj* that represents a context-insensitive field-based analysis. L_C enforces the part of *k-obj* that is object-sensitive for both method calls and field accesses (by making the field-based analysis defined in L_F also field-sensitive (Figure 8)). When computing the

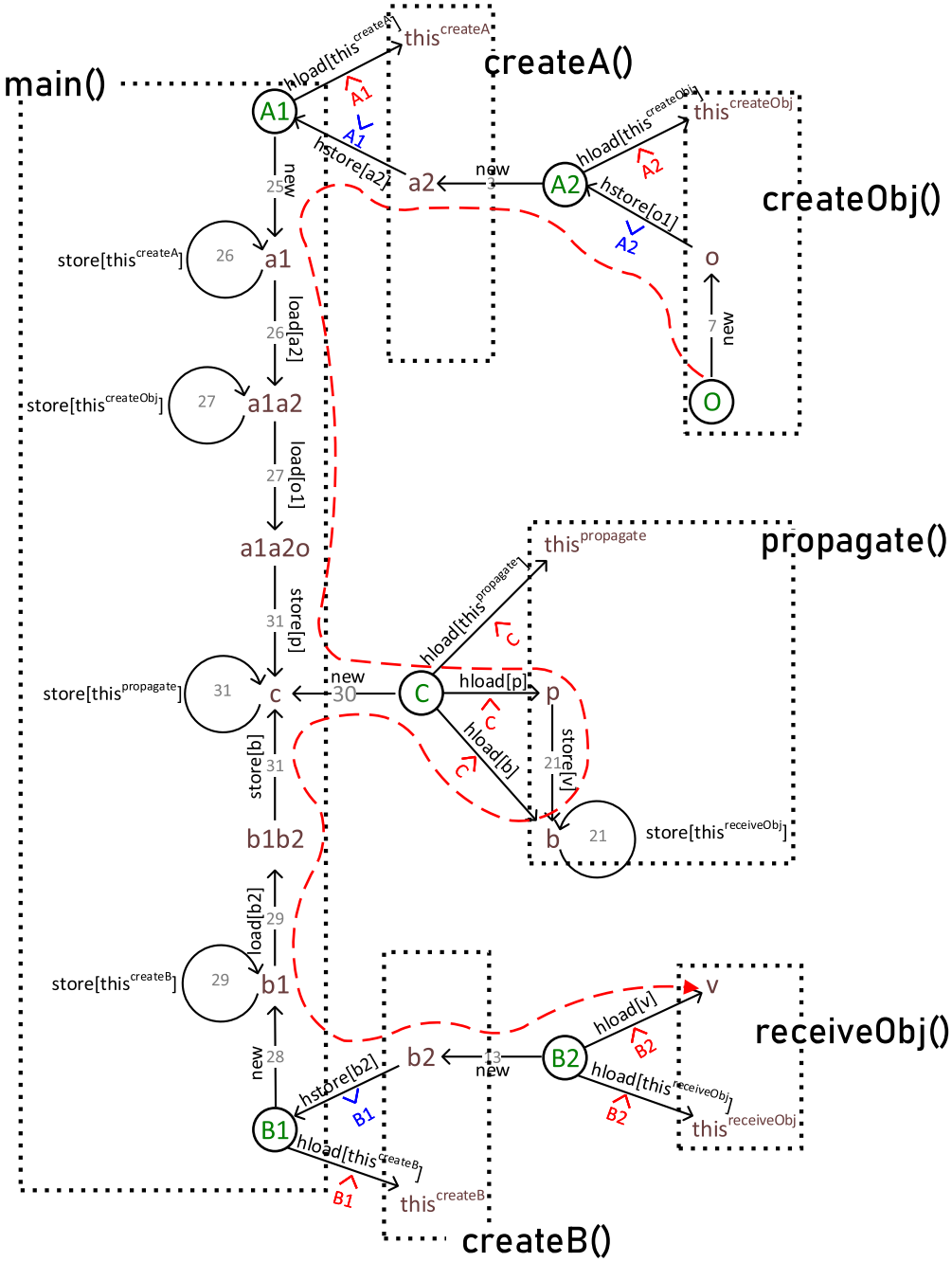


Fig. 12. The PAG for the program given in Figure 11 (with the L_{FC}-path from 0 to v highlighted using a dashed line).

points-to information under k-limiting, *CFL-obj* will simply maintain a context stack of length k, which always keeps track of the k mostly recent context elements pushed to the context stack (as is standard). In addition, G_{pag} is a super-graph of the PAG that is built by *CFL-obj*, as G_{pag} is pre-computed by a less precise pointer analysis. By computing the points-to information over G_{pag} according to $L_{FC} = L_F \cap L_C$, *CFL-obj* under k-limiting will thus be equivalent to k-*obj*, as both will produce exactly the same points-to information. □

3.4 Precision-Preserving CFL-Reachability-Based Pre-Analysis

We describe our CFL-reachability-based pre-analysis to enable k-*obj* to perform its pointer analysis with partial context sensitivity. As illustrated in Figure 2, our pre-analysis determines if a variable/allocation site should be analyzed by k-*obj* context-sensitively or not by checking if a CFL-reachability-based condition holds or not. For a given program, we will apply our pre-analysis to the program only once and then make such selections for k-*obj* identically for all values of k to enable its faster analysis for the same program during a subsequent main analysis. To preserve precision, as illustrated in Figure 5, our pre-analysis must be conservative in order not to miss any value flow in the program.

Given a program, our precision-preserving pre-analysis, which is developed by solving a reachability problem in $L_{FC} = L_F \cap L_C$ over-approximately, operates on a pre-built PAG for the program by the analysis of Andersen [1] as a fully context-sensitive taint analysis so that it can not only maximize effectively the number of variables/allocation sites that will be analyzed by k-*obj* context-insensitively in the program but also run linearly in terms of the edges in the PAG. Thus, when over-approximating $L_{FC} = L_F \cap L_C$, we will simplify L_F (by regularizing it) while keeping L_C unchanged.

To ease understanding, we describe the development of our pre-analysis incrementally in three stages. In Section 3.4.1, we start with a pre-analysis for solving a reachability problem in $L_{FC} = L_F \cap L_C$. In Section 3.4.2, we examine a second pre-analysis for solving a reachability problem in $L_{OC} = L_O \cap L_C$, where L_O is a superset of L_F developed based on a new object-based abstraction. In Section 3.4.3, we introduce a third pre-analysis, which is our final solution, for solving a reachability problem in $L_{RC} = L_R \cap L_C$ as a taint analysis (without actually computing the points-to information), where L_R is regularized from L_O so that L_R is a superset of L_O.

3.4.1 $L_{FC} = L_F \cap L_C$. Initially, we consider a pre-analysis, FC_{pre}, for solving (conceptually) an L_{FC}-reachability problem (without k-limiting). For a program, let its G_{pag} be pre-computed by the analysis of Andersen [1]. Given a path from O to w in G_{pag}, we know that w points to O iff it is both an L_F-path and an L_C-path. We can recast CFL-P1 ∧ CFL-P2 in Figure 2 with respect to $L_{FC} = L_F \cap L_C$ easily as follows. A variable/allocation site n (in a method m) will be selected to be context-sensitive if there exists a path $p_{O,n,w}$ in G_{pag}, formed by two subpaths, one from some O to n, denoted $p_{O,n}$, and one from n to some w, denoted $p_{n,w}$, such that FC-P1 ∧ FC-P2 ∧ FC-P3 holds:

FC-P1 (*O*, *n*, *w*): $L_C(p_{O,n}) \in L_C$, where $ctx(L_C^{entry}(p_{O,n})) \neq [\,]$;
FC-P2 (*O*, *n*, *w*): $L_C(p_{n,w}) \in L_C$, where $ctx(L_C^{exit}(p_{n,w})) \neq [\,]$; and
FC-P3 (*O*, *n*, *w*): $L_F(p_{o,n,w}) \in L_F$.

Here, FC-P1 ensures that the value-flow path $p_{O,n}$ is realizable (when $L_C(p_{O,n}) \in L_C$) and inter-procedural (when $ctx(L_C^{entry}(p_{O,n})) \neq [\,]$, i.e, $p_{O,n}$ enters m under a nonempty context $ctx(L_C^{entry}(p_{O,n}))$). This means that O will flow into $n.f_1.\ldots.f_{r_f}$ for some $f_1.\ldots.f_{r_f}$ identified in $p_{O,n}$. FC-P2 can be understood similarly, ensuring that $n.g_1.\ldots.g_{r_g}$ will flow into w along $p_{n,w}$ inter-procedurally. Finally, FC-P3 ensures that $p_{O,n,w}$ is a (context-insensitive) *flowsto* path in L_F,

which implies that $f_1.....f_{r_f}$ and $g_1.....g_{r_g}$ are identical, so that $n.f_1.....f_{r_f}$ and $n.g_1.....g_{r_g}$ are aliases in L_F. Note that FC_{pre} does not need to explicitly compute such aliases as they are enforced implicitly (by FC-P3).

Let us verify this condition to see why the variable r in Figure 4 must be context-sensitive, by considering the path consisting of the subpath from 01 to f and the subpath from f to w2 in its PAG given in Figure 10:

$$P_{01,r,w2} \overset{\text{def}}{=\!=} 01 \xrightarrow{new} o1 \xrightarrow{store[f]} b1 \xrightarrow[\widehat{A1}]{\overline{load[r]}} a1 \xrightarrow{\overline{new}} A1 \xrightarrow[\widehat{A1}]{\overline{hstore[r]}} r \xrightarrow{\overline{new}} B \xrightarrow[\widehat{B}]{hload[f]} f$$
$$\xrightarrow[\check{B}]{hstore[f]} B \xrightarrow{new} r \xrightarrow[\check{A2}]{hstore[r]} A2 \xrightarrow{new} a2 \xrightarrow{load[r]} b2 \xrightarrow{load[f]} w2 \quad . \tag{13}$$

Note that this path is $P1 + Q2$ given in (7) and (8) but merged at node f. We can now establish CFL-P1 : $01 \underset{[A1]}{\Longrightarrow} r.f$ and CFL-P2 : $r.f \underset{[A2]}{\Longrightarrow} w2$ as highlighted in Figure 10 with CFL-reachability. For $P_{01,r,w2}$, we can chop it into two subpaths, $P_{01,r}$ and $P_{r,w2}$, at the first or second occurrence of r. FC-P1 through FC-P3 are all satisfied, since $L_C(P_{01,r}) = \widehat{A1} \in L_C$, where $ctx(L_C^{entry}([\widehat{A1}]) = [A1] \neq [\,]$, $L_C(P_{r,w2}) = \check{A2} \in L_C$, where $ctx(L_C^{exit}([\check{A2}]) = [A2] \neq [\,]$, and $L_F(P_{01,r,w2}) \in L_F$ holds trivially. Similarly, we can verify that B must also be context-sensitive (by substituting B for r in the preceding reasoning).

According to the following theorem, we can at least solve $L_{FC} = L_F \cap L_C$ (conceptually but not in actuality) as a pre-analysis to accelerate *k-obj* with partial context sensitivity while preserving its precision. We first prove the theorem and then illustrate it with an example. It is understood that null that can be modeled as a special object, say, nullobj, during a CFL-reachability-based pointer analysis (even performed flow-insensitively).

Theorem 3.2 [Precision for L_{FC}]. *Let FC_{pre} be the pre-analysis for solving a reachability problem in $L_{FC} = L_F \cap L_C$ (without k-limiting) for a program P, operating on its G_{pag} pre-computed by the analysis of Andersen [1]. A variable/allocation site in a method in G_{pag} is selected to be context-sensitive according to FC-P1 through FC-P3. Let k-obj_{FC} be the resulting version of k-*obj*. Then k-obj_{FC} has exactly the same precision as k-*obj*: $\overline{pt}_{k\text{-}obj_{FC}}(v) = \overline{pt}_{k\text{-}obj}(v)$ for every variable v in P.*

Proof. Let us consider a generic variable/allocation site n in a generic method m in P's G_{pag}. Let us assume that FC-P1 $\wedge$ FC-P2 $\wedge$ FC-P3 does not hold for n. Then *k-obj*$_{FC}$ will choose to analyze n context-insensitively but *k-obj* will analyze n context-sensitively instead. We claim that these two different choices will not make any difference in terms of the points-to information computed for P. In the following, we prove this claim by contradiction. Suppose that *k-obj* is more precise than *k-obj*$_{FC}$. Thus, if m is reachable (i.e., analyzed by *k-obj*), then m will also be reachable by *k-obj*$_{FC}$. By Theorem 3.1, it suffices to prove the claim by assuming that m is reachable under *k-obj* (and thus *k-obj*$_{FC}$), since then both *k-obj* and *k-obj*$_{FC}$ will be known to analyze exactly the same methods in G_{pag} with the same precision.

If *k-obj* is more precise than *k-obj*$_{FC}$, then there must exist two context-sensitive value-flow paths in P passing through n, identified as p_{O_1,n,w_1} and p_{O_2,n,w_2}, in G_{pag}, such that (C1) $O_1 \underset{c_1}{\Longrightarrow} n.f_1.....f_r \wedge n.f_1.....f_r \underset{c_1}{\Longrightarrow} w_1$ under a nonempty context c_1 of m and (C2) $O_2 \underset{c_2}{\Longrightarrow} n.f_1.....f_r \wedge n.f_1.....f_r \underset{c_2}{\Longrightarrow} w_2$ under a nonempty context c_2 of m, where $O_1 \neq O_2$, $w_1 \neq w_2$ and $c_1 \neq c_2$. As far as p_{O_1,n,w_1} and p_{O_2,n,w_2} are concerned, *k-obj*$_{FC}$ will conclude that $\{O_1, O_2\} \subseteq \overline{pt}_{k\text{-}obj_{FC}}(w_1)$ and $\{O_1, O_2\} \subseteq \overline{pt}_{k\text{-}obj_{FC}}(w_2)$ since it cannot separate the two value flows when analyzing n context-insensitively. In contrast, *k-obj* can separate the two value flows as it analyzes n context-sensitively and will thus be more precise by finding that $O_1 \in \overline{pt}_{k\text{-}obj}(w_1)$ and $O_2 \in \overline{pt}_{k\text{-}obj}(w_2)$

```
1  class B {
2      Object id(Object p) {
3          return p;
4      }
5  }
6  class A {
7      B create() {
8          B r = new B(); // B
9          return r;
10     }
11 }
12 class C {
13     Object foo(Object x) {
14         A a1 = new A(); // A1
15         A a2 = new A(); // A2
16         B b1 = a1.create();
17         B b2 = a2.create();
18         Object v1 = b1.id(x);
19         Object y = new Object(); // O3
20         Object v2 = b2.id(y);
21         return v2
22     }
23 }
24 main() {
25     C c1 = new C(); // C1
26     C c2 = new C(); // C2
27     Object o1 = new Object(); // O1
28     Object o2 = new Object(); // O2
29     Object w1 = c1.foo(o1);
30     Object w2 = c2.foo(o2);
31 }
```

Fig. 13. An example for illustrating Theorem 3.2.

but $O_2 \notin \overline{pt}_{k\text{-}obj}(w_1)$ and $O_1 \notin \overline{pt}_{k\text{-}obj}(w_2)$. However, this outcome cannot be possible. For any of the two conditions Ci, where $1 \leqslant i \leqslant 2$, stated earlier, we are given that FC-P1$(O_i, n, w_i) \wedge$ FC-P2(O_i, n, w_i) holds, which implies that FC-P3(O_i, n, w_i) holds trivially. Thus, FC-P1$(O_i, n, w_i) \wedge$ FC-P2$(O_i, n, w_i) \wedge$ FC-P3(O_i, n, w_i) holds, but this contradicts with the assumption made about the falsehood of this condition at the beginning of our proof by contradiction. This implies that k-*obj* must behave identically as k-obj_{FC} for w_1 and w_2 so that $\{O_1, O_2\} \subseteq \overline{pt}_{k\text{-}obj}(w_1)$ and $\{O_1, O_2\} \subseteq \overline{pt}_{k\text{-}obj}(w_2)$. Thus, k-obj_{FC} and k-*obj* yield exactly the same points-to set for every variable v in P: $\overline{pt}_{k\text{-}obj_{FC}}(v) = \overline{pt}_{k\text{-}obj}(v)$. □

In the following, we illustrate Theorem 3.2 with an example given in Figure 13. For this program, G_{pag} (not shown) includes all methods in the program. For all variables/allocation sites in the program, `p`, `r`, and `B` are the only three requiring context sensitivity in k-*obj* (for any value of k) according to FC_{pre}. Let us see how k-*obj* and k-obj_{FC} are equivalent by focusing on the points-to information computed for `w1` and `w2`. For this example, we need only to consider $k \in \{1, 2\}$:

- 1-***obj*** **vs.** 1-$\boldsymbol{obj}_{FC}$: Under 1-obj_{FC}, we have $\overline{pt}_{1\text{-}obj_{FC}}(w_1) = \overline{pt}_{1\text{-}obj_{FC}}(w_2) = \{\texttt{O1}, \texttt{O2}, \texttt{O3}\}$. Under *1-obj*, which is not powerful enough to distinguish the two calls, `b1.id(x)` and `b2.id(y)`, to `id()`, since both `b1` and `b2` are considered to point to the same object, `B`, *1-obj* will obtain the same points-to sets: $\overline{pt}_{1\text{-}obj}(w_1) = \overline{pt}_{1\text{-}obj}(w_2) = \{\texttt{O1}, \texttt{O2}, \texttt{O3}\}$.
- 2-***obj*** **vs.** 2-$\boldsymbol{obj}_{FC}$: For this program, 2-*obj* is more precise than *1-obj*. By recognizing that `b1` points to ([`A1`], `B`) and `b2` points to ([`A2`], `B`), *2-obj* can now analyze `id()` separately for the two calls to `id()` without conflating these two different objects as in *1-obj*. Thus, $\overline{pt}_{2\text{-}obj}(w_1) = \overline{pt}_{2\text{-}obj}(w_2) = \{\texttt{O3}\}$. Under 2-$obj_{FC}$, `b1` and `b2` are context-insensitive but `B` is context-sensitive. Thus, `b1` and `b2` also point to ([`A1`], `B`) and ([`A2`], `B`), respectively. By distinguishing also the two calls to `id()` as *2-obj* does but analyzing `v2` context-insensitively (where `v2` points to `O3`), 2-obj_{FC} will also obtain the same points-to sets for `w1` and `w2`: $\overline{pt}_{2\text{-}obj_{FC}}(w_1) = \overline{pt}_{2\text{-}obj_{FC}}(w_2) = \{\texttt{O3}\}$.

3.4.2 $L_{OC} = L_O \cap L_C$. Now, we consider a pre-analysis, OC_{pre}, for solving an L_{OC}-reachability problem (without k-limiting), where L_O is over-approximated to be a superset of L_F under a so-called new *object-based abstraction*. This novel abstraction represents a significant intermediate step for enabling us to eventually obtain a pre-analysis that is both effective (by operating fully context-sensitively to reduce signifiantly the number of context-sensitive facts inferred by k-*obj*) and efficient (by operating as a taint analysis linearly). As for G_{pag} for a program, OC_{pre}

again operate on the PAG pre-computed for the program over-approximately by the analysis of Andersen [1].

Under our object-based abstraction, we will apply 1-limiting to a load $\cdots = v.g$ (by abstracting $v.g$ with v) unconditionally and 1-limiting to a store $v.f = \cdots$ (by abstracting $v.f$ with v), where v points to an object O, only when there is also a store-load pair, $O.f' = \cdots$ and $\cdots = O.f'$, where f and f' are not necessarily different. This is more precise than 1-limited access paths for object sensitivity. Recall that in our PAG representation, the parameters $p_i^{m'}$ of a method m' and the local variable $ret^{m'}$ for storing the return values of m' are modeled as fields of its receivers. According to L_F in (4), if $p_i^{m'}$ *flows* $ret^{m'}$, then $p_i^{m'}$ and $ret^{m'}$ are regarded as the same field of a receiver object of m' (Section 3.3).

The grammar for defining L_O (over-approximately as an superset of L_F) is given next:

$$\begin{array}{rcl} \mathit{flowsto} & \rightarrow & \mathsf{new}\ \mathit{flows}^* \\ \overline{\mathit{flowsto}} & \rightarrow & \overline{\mathit{flows}}^*\ \overline{\mathsf{new}} \\ \mathit{flows} & \rightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathit{flowsto}}\ \mathsf{hload} \mid \mathsf{hstore}\ \mathsf{new} \mid \overline{\mathsf{load}}. \\ \overline{\mathit{flows}} & \rightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{hload}}\ \mathit{flowsto}\ \overline{\mathsf{store}} \mid \overline{\mathsf{new}}\ \overline{\mathsf{hstore}} \mid \overline{\mathsf{load}} \end{array} \tag{14}$$

Unlike L_F, L_O is no longer field-sensitive. Thus, the number of terminals in {new, $\overline{\texttt{new}}$, assign, $\overline{\texttt{assign}}$, store, $\overline{\texttt{store}}$, hload, $\overline{\texttt{hload}}$, hstore, $\overline{\texttt{hstore}}$, load, $\overline{\texttt{load}}$} in the grammar for L_O is 12, which is significantly less than that in the grammar for L_F. Like L_F, however, L_O remains to be a balanced-parentheses language, except it requires only a store edge to be matched by an hload edge and an $\overline{\texttt{hload}}$ edge to be matched by a $\overline{\texttt{store}}$ edge (field-insensitively).

Note that even though L_O is field-insensitive. The fields of objects (including the parameters/return variables of methods) are modeled explicitly in our object-sensitive PAG representation, as can be observed in Figures 9, 10, and 12.

Lemma 3.3 [Inclusion]. $L_O \supseteq L_F$.

Proof. To move from L_F to L_O, the following productions in the grammar for L_F given in (4),

$$\begin{array}{rcl} \mathit{flows} & \rightarrow & \mathsf{assign} \mid \mathsf{store}[f]\ \overline{\mathit{flowsto}}\ \mathsf{hload}[f] \mid \mathsf{hstore}[f]\ \mathit{flowsto}\ \mathsf{load}[f] \\ \overline{\mathit{flows}} & \rightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{hload}[f]}\ \mathit{flowsto}\ \overline{\mathsf{store}[f]} \mid \overline{\mathsf{load}[f]}\ \overline{\mathit{flowsto}}\ \overline{\mathsf{hstore}[f]}, \end{array} \tag{15}$$

are first over-approximated (by making the underlying language field-insensitive with f and g to be matchable):

$$\begin{array}{rcl} \mathit{flows} & \rightarrow & \mathsf{assign} \mid \mathsf{store}[f]\ \overline{\mathit{flowsto}}\ \mathsf{hload}[g] \mid \mathsf{hstore}[f]\ \mathit{flowsto}\ \mathsf{load}[g] \\ \overline{\mathit{flows}} & \rightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{hload}[f]}\ \mathit{flowsto}\ \overline{\mathsf{store}[g]} \mid \overline{\mathsf{load}[f]}\ \overline{\mathit{flowsto}}\ \overline{\mathsf{hstore}[g]}, \end{array} \tag{16}$$

that is,

$$\begin{array}{rcl} \mathit{flows} & \rightarrow & \mathsf{assign} \mid \mathsf{store}\ \overline{\mathit{flowsto}}\ \mathsf{hload} \mid \mathsf{hstore}\ \mathit{flowsto}\ \mathsf{load} \\ \overline{\mathit{flows}} & \rightarrow & \overline{\mathsf{assign}} \mid \overline{\mathsf{hload}}\ \mathit{flowsto}\ \overline{\mathsf{store}} \mid \overline{\mathsf{load}}\ \overline{\mathit{flowsto}}\ \overline{\mathsf{hstore}}, \end{array} \tag{17}$$

which is then over-approximated again to yield the productions used for defining L_O given in (14). Therefore, for every object O and every variable v in the program, if O *flowsto* v in L_F, then O *flowsto* v in L_O. □

We discuss in the following how aliases are over-approximated under our object-based abstraction in L_O to facilitate its understanding (e.g., about its differences with 1-limited access paths). Note that aliases are implicitly recognized and handled during our CFL-reachability-based pre-analysis. If $w = n.g$ and $n.f = v$ exist in a program P, then $n.g$ and $n.f$ are aliases in L_O such that $w = v$ iff there exists a load $\cdots = n'.f$, where n and n' point to a common object, say, O:

- For the load $w = n.g$, its edge $n \xrightarrow{\texttt{load[g]}} w$ (added to G_{pag} by [Load]) is simplified in L_O to $n \xrightarrow{\texttt{load}} w$, which is treated unconditionally (with 1-limiting to $n.g$) as an $n \xrightarrow{\texttt{assign}} w$ edge. Thus, $w = n.g$ is modeled as $w = n$.
- For the store $n.f = v$, where n points to O, $v \xrightarrow{\texttt{store[f]}} n$ and $O \xrightarrow{\texttt{hload[f]}} f$ are added to G_{pag} by [Store]. In L_O, we must have $v \xrightarrow{\texttt{store}} n\ \overline{flowsto}\ O \xrightarrow{\texttt{hload}} f\ \boxed{\cdots}\ O\ flowsto\ n$. In L_O, $n.f = v$ is treated as $n = v$ (with 1-limiting to $n.f$) iff there exists a load $\cdots = n'.f$ in P, where n' also points to O (in L_O), so that it can enable an $f \xrightarrow{\texttt{hstore[f]}} O$ edge to be added in G_{pag}. When this happens, the $\cdots$ above can be replaced by $f \xrightarrow{\texttt{hstore}} O$ so that $v \xrightarrow{\texttt{store}} n\ \overline{flowsto}\ O \xrightarrow{\texttt{hload}} f \xrightarrow{\texttt{hstore}} O\ flowsto\ n$, which implies that $n.f = v$ has been modeled as $n = v$.

An analogue of FC-P1 through FC-P3 given earlier for FC_{pre} is stated for OC_{pre} next (with the difference marked in bold):

OC-P1 (O, n, w): $L_C(p_{O,n}) \in L_C$, where $ctx(L_C^{entry}(p_{O,n})) \neq [\,]$;
OC-P1 (O, n, w): $L_C(p_{n,w}) \in L_C$, where $ctx(L_C^{exit}(p_{n,w})) \neq [\,]$; and
OC-P3 (O, n, w): $L_O(p_{o,n,w}) \in \boldsymbol{L_O}$.

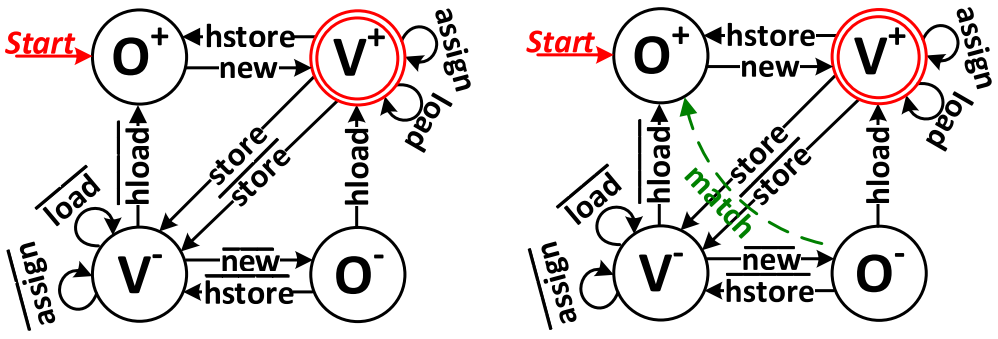
Here, OC-P1 ensures that the value-flow path $p_{O,n}$ is realizable (when $L_C(p_{O,n}) \in L_C$) and inter-procedural (when $ctx(L_C^{entry}(p_{O,n})) \neq [\,]$ i.e, $p_{O,n}$ enters m under a nonempty context $ctx(L_C^{entry}(p_{O,n}))$). This means that O will flow into $n.f_1.\ldots.f_{r_f}$ for some $f_1.\ldots.f_{r_f}$ identified in $p_{O,n}$. OC-P2 can be understood similarly, ensuring that $n.g_1.\ldots.g_{r_g}$ will flow into w along $p_{n,w}$ inter-procedurally. OC-P3 ensures that $p_{O,n,w}$ is a (context-insensitive) *flowsto* path in L_O, which implies that $n.f_1.\ldots.f_{r_f}$ and $n.g_1.\ldots.g_{r_g}$ are aliases under our object-based abstraction in L_O.

The following theorem stated for OC_{pre} is an analogue of Theorem 3.2 stated earlier for FC_{pre}. If FC_{pre} is precision-preserving, then OC_{pre}, which is less precise, must also be precision-preserving.

Theorem 3.4 [Precision for L_{OC}]. *Let OC_{pre} be the pre-analysis for solving a reachability problem in $L_{OC} = L_O \cap L_C$ (without k-limiting) for a program P, operating on its G_{pag} pre-computed by the analysis of Andersen [1]. A variable/allocation site in a method in G_{pag} is selected to be context-sensitive according to OC-P1–OC-P3. Let k-obj_{OC} be the resulting version of k-obj. Then k-obj_{OC} has exactly the same precision as k-obj: $\overline{pt}_{k\text{-}obj_{OC}}(v) = \overline{pt}_{k\text{-}obj}(v)$ for every variable v in P.*

Proof. Let CI_{OC} (CI_{FC}) be the set of variables/allocation sites in P that are selected to be analyzed by k-*obj* context-insensitively by OC_{pre} (FC_{pre}). As $L_O \supseteq L_F$ (Lemma 3.3), which implies that $L_{OC} \supseteq L_{FC}$, we must have $CI_{OC} \subseteq CI_{FC}$ (as OC_{pre} is more conservative than FC_{pre}). Finally, a further application of Theorem 3.2 concludes the proof. □

Given $L_{OC} = L_O \cap L_C$, the pre-analysis OC_{pre} for computing its CFL-reachability information is still undecidable in general [32] and only polynomially solvable under k-limiting. In the latter case, the worst-time complexity for finding the points-to set of a variable is $O(N^3_{OC_{\text{pag}}} \Gamma^3_{L_{OC}})$, where



(a) L_R as a DFA (b) $L_R^{\texttt{match}}$ with a match transition

Fig. 14. Regularizing L_O as L_R and exploiting a match transition (added to L_R) to recognize balanced L_{RC}-subpaths stacklessly.

$N_{OC_{\text{pag}}}$ is the number of nodes in $G_{OC_{\text{pag}}}$ (the PAG finally discovered by OC_{pre}) and $\Gamma_{L_{OC}}$ is the size of L_{OC} [14, 31]. In comparison with the worst-case complexity $O(N^3_{FC_{\text{pag}}}\Gamma^3_{L_{FC}})$ for FC_{pre} under k-limiting (Section 3.4.1), $N^3_{OC_{\text{pag}}}$ is usually larger than $N^3_{FC_{\text{pag}}}$ but $\Gamma^3_{L_{OC}}$ is much smaller than $\Gamma^3_{L_{FC}}$.

3.4.3 $L_{RC} = L_R \cap L_C$. Finally, we present our pre-analysis, Eagle, for solving a reachability problem in $L_{RC} = L_R \cap L_C$ *without k-limiting*, where L_R is a specially crafted superset of L_O, so that Eagle is both effective in enabling certain variables/allocation sites to be analyzed by *k-obj* context-insensitively so as to reduce the number of context-sensitive facts inferred by *k-obj* in the program (since L_C is fully context-sensitive) and efficient (since L_{RC} will be solved linearly in terms of the edges in a PAG of the program). Unlike FC_{pre} or OC_{pre}, Eagle will be formulated as a fully context-sensitive taint analysis (without the need to compute any points-to information).

To make this possible, we will turn $L_{OC} = L_O \cap L_C$ into $L_{RC} = L_R \cap L_C$ with three transformations:

1. **Using a pre-computed PAG G_{pag}:** For a given program, Eagle will operate on G_{pag}, a PAG pre-built by using the points-to information pre-computed by the analysis of Andersen [1] (context-insensitively). By construction, G_{pag} is a supergraph of the PAG built by *k-obj* for any value of k (where $k \geqslant 1$). Without actually computing any points-to information, Eagle will directly perform its taint analysis on G_{pag} conservatively.
2. **Regularizing L_O to L_R:** L_R, which is given as a DFA shown in Figure 14(a), computes L_O in (14) over-approximately. There are four states: O^+ (O^-) signifies that the DFA has just started (finished) in computing a *flowsto* ($\overline{flowsto}$) relation for an object O, and V^+ (V^-) indicates that the DFA is computing a *flowsto* ($\overline{flowsto}$) for a variable V. Therefore, this DFA computes *flowsto* when starting from O^+ and terminating at V^+ and $\overline{flowsto}$ (i.e., the standard points-to relation when starting from V^- and terminating at O^-).

   According to L_R, every node n in G_{pag} can be in one of the two states: n^+ (participating in computing *flowsto*) and n^- (participating in computing $\overline{flowsto}$). We write $G_{\text{R-pag}}$ to represent a *regularized PAG* of G_{pag}, obtained from G_{pag} with the source and target nodes of each of its regular or inverse edges being labeled explicitly by their states according

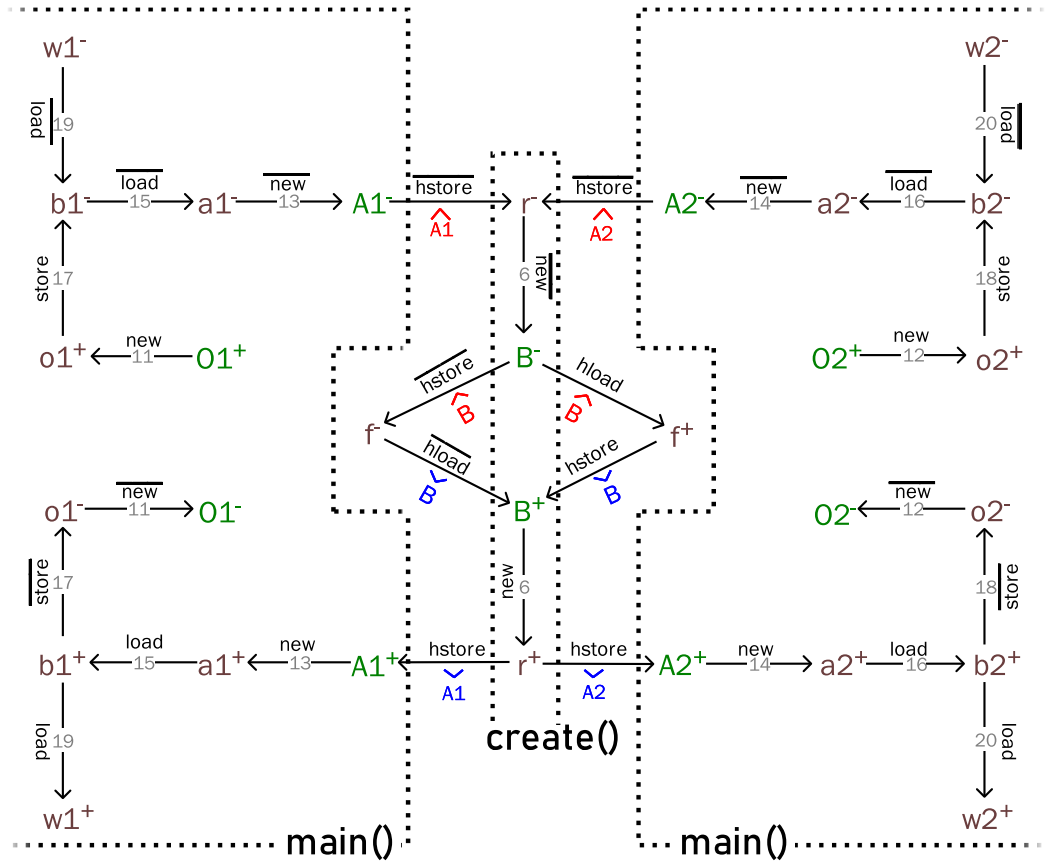


Fig. 15. The PAG regularized from Figure 10 with all inverse edges shown explicitly. To avoid cluttering, each node n is shown as two separate nodes, n^+ and n^-, which represent n in two different states according to the DFA in Figure 14(a).

to L_R. For each edge $x^s \xrightarrow[O]{\ell} y^t$ in $G_{\text{R-pag}}$, where $s, t \in \{+, -\}$, its inverse edge is $y^{\overline{t}} \xrightarrow[\overline{O}]{\overline{\ell}} x^{\overline{s}}$ such that $\overline{+} = -$ and $\overline{-} = +$. Figure 15 gives the regularized version of the PAG depicted in Figure 10. In this regularized PAG, $\mathsf{B} \xrightarrow[\widehat{\mathsf{B}}]{\mathtt{hload}} \mathsf{f}$ becomes $\mathsf{B}^- \xrightarrow[\widehat{\mathsf{B}}]{\mathtt{hload}} \mathsf{f}^+$ and $\mathsf{f} \xrightarrow[\check{\mathsf{B}}]{\overline{\mathtt{hload}}} \mathsf{B}$ becomes $\mathsf{f}^- \xrightarrow[\check{\mathsf{B}}]{\overline{\mathtt{hload}}} \mathsf{B}^+$, for example.

In L_R, the allocation sites (i.e., objects) and variables (including fields as a special cases) are no longer distinguished. For any two nodes $n_1^{s_1}$ and $n_2^{s_2}$ in $G_{\text{R-pag}}$, we write $n_1^{s_1} \xrightarrow{L_R} n_2^{s_2}$ if L_R, when starting at $n_1^{s_1}$, can move to $n_2^{s_2}$. According to L_R, $n_1^{s_1} \xrightarrow{L_R} n_2^{s_2}$ iff $n_2^{\overline{s_2}} \xrightarrow{L_R} n_1^{\overline{s_1}}$. If $n_1^+ \xrightarrow{L_R} n_2^+$, whatever n_1 points to flows to n_2. Conversely, iff $n_2^- \xrightarrow{L_R} n_1^-$, n_2 points to whatever to n_1 points to. Thus, O *flowsto* v (v $\overline{\textit{flowsto}}$ O) in L_R iff $O^+ \xrightarrow{L_R} v^+$ ($v^- \xrightarrow{L_R} O^-$).

Lemma 3.5 [Inclusion]. $L_R \supseteq L_O$.

Proof. L_R, which is specified by the DFA in Figure 14(a), can also be specified equivalently as a regular language. One possible regular grammar is given next:

$$\begin{array}{rcl}
\mathit{flowsto} & \rightarrow & \texttt{new}\ \mathit{flows}^* \\
\overline{\mathit{flowsto}} & \rightarrow & \overline{\mathit{flows}}^*\ \overline{\texttt{new}} \\
\mathit{flows} & \rightarrow & \texttt{assign} \\
 & | & \texttt{load} \\
 & | & (\texttt{store} \mid \overline{\texttt{store}})\ (\overline{\texttt{assign}} \mid \overline{\texttt{load}} \mid \overline{\texttt{new}}\ \overline{\texttt{hstore}})^*(\overline{\texttt{new}}\ \texttt{hload} \mid \overline{\texttt{hload}}\ \texttt{new}). \\
 & | & \texttt{hstore new} \\
\overline{\mathit{flows}} & \rightarrow & \overline{\texttt{assign}} \\
 & | & \overline{\texttt{load}} \\
 & | & (\overline{\texttt{new}}\ \texttt{hload} \mid \overline{\texttt{hload}}\ \texttt{new})(\texttt{assign} \mid \texttt{load} \mid \texttt{hstore new}\ )^*\ (\texttt{store} \mid \overline{\texttt{store}}) \\
 & | & \overline{\texttt{new}}\ \overline{\texttt{hstore}}
\end{array} \tag{18}$$

Given the regular language defined earlier, we can proceed similarly as in the proof of Lemma 3.3 to show that for every object O and every variable v, if O *flowsto* v in L_O, then O *flowsto* v in L_R. □

If we run L_R on the regularized PAG in Figure 15 starting from 01/02, we will find that $\texttt{01}^+/\texttt{02}^+ \xrightarrow{L_R} \texttt{w1}^+/\texttt{w2}^+$ (i.e., 01/02 *flowsto* w1/w2 in L_R). For example, $\texttt{01}^+ \xrightarrow{L_R} \texttt{w2}^+$ (i.e., 01 *flowsto* w2) is established as follows:

$$\begin{array}{l}
\texttt{01}^+ \xrightarrow{\texttt{new}} \texttt{o1}^+ \xrightarrow{\texttt{store}} \texttt{b1}^- \xrightarrow{\overline{\texttt{load}}} \texttt{a1}^- \xrightarrow{\overline{\texttt{new}}} \texttt{A1}^- \xrightarrow[\widehat{\texttt{A1}}]{\overline{\texttt{hstore}}} \texttt{r}^- \xrightarrow{\overline{\texttt{new}}} \texttt{B}^- \xrightarrow[\widehat{\texttt{B}}]{\texttt{hload}} \texttt{f}^+ \\
\xrightarrow[\check{\texttt{B}}]{\texttt{hstore}} \texttt{B}^+ \xrightarrow{\texttt{new}} \texttt{r}^+ \xrightarrow[\check{\texttt{A2}}]{\texttt{hstore}} \texttt{A2}^+ \xrightarrow{\texttt{new}} \texttt{a2}^+ \xrightarrow{\texttt{load[r]}} \texttt{b2}^+ \xrightarrow{\texttt{load}} \texttt{w2}^+
\end{array} . \tag{19}$$

If L_C is also considered, however, only 01 *flowsto* w1 and 02 *flowsto* w2 hold.

3. **Recognizing balanced L_{RC}-*subpaths stacklessly*:** To account for L_C, we may be tempted to design a pre-analysis, RC_{pre}, for verifying an analogue of OC-P1 through OC-P3 given earlier for OC_{pre}:

   RC-P1 (*O*, *n*, *w*): $L_C(p_{O,n}) \in L_C$, where $ctx(L_C^{entry}(p_{O,n})) \neq [\,]$;
   RC-P1 (*O*, *n*, *w*): $L_C(p_{n,w}) \in L_C$, where $ctx(L_C^{exit}(p_{n,w})) \neq [\,]$; and
   RC-P3 (*O*, *n*, *w*): $L_R(p_{o,n,w}) \in \boldsymbol{L_R}$.

   However, this would require us to use a context stack to match explicitly all balanced parentheses identified according to L_C as has been done traditionally [5, 33, 37, 41, 45]. As a result, RC_{pre} would be undecidable in general [32] and still only polynomially solvable even with k-limiting [14, 31].

   Instead, Eagle is designed to solve $L_{RC} = L_R \cap L_C$ fully context-sensitively without k-limiting in linear time (in terms of the number of edges in $G_{\text{R-pag}}$ and some `match` edges introduced shortly in the following).

   To make Eagle linear, the key is to recognize L_C stacklessly by leveraging the following three important insights:

   - First, as L_R is regular, we can establish RC-P1 through RC-P3 equivalently by solving a fully context-sensitive CFL-reachability-based taint analysis in $G_{\text{R-pag}}$ without having to compute any points-to information explicitly.

     In L_R, which is regular, we can reason about CFL-reachability by simply tracking its state transitions. For a method m, let each of its node, n^s, in $G_{\text{R-pag}}$, where $s \in \{+, -\}$, be associated with an attribute, cs, such that n^s is tainted iff $n^s.cs = \text{true}$.

EAGLE will set $n^+.cs$ = true iff $O^+ \xrightarrow{L_R} n^+$ *holds inter-procedurally* for m in L_C, indicating that some object O flows into n from outside m. EAGLE will set $n^-.cs$ = true iff $w^- \xrightarrow{L_R} n^-$ *holds inter-procedurally* for m in L_C, indicating that whatever n points-to is also pointed to by some variable w from outside m. Thus, a variable/allocation site n in a method m is context-sensitive if EAGLE-P1 ∧ EAGLE-P2 holds:

EAGLE-P1(n): $n^+.cs$ = true; and
EAGLE-P2(n): $n^-.cs$ = true.

Note that RC-P3 stated earlier is implied, since if EAGLE-P1 ∧ EAGLE-P2 holds, then $O^+ \xrightarrow{L_R} n^+ \wedge w^- \xrightarrow{L_R} n^-$ (i.e., $O^+ \xrightarrow{L_R} n^+ \wedge n^+ \xrightarrow{L_R} w^+$ holds), implying that the path starting from O and ending at w via n is a *flowsto* path in L_R (i.e., RC-P3 holds). (As EAGLE-P1 ensures that CFL-P1 : $O \underset{c_1}{\Longrightarrow} n.f_1.\ldots.f_{r_f}$ and EAGLE-P2 ensures that CFL-P2 : $n.g_1.\ldots.g_{r_g} \underset{c_2}{\Longrightarrow} w$, both ensure that $n.f_1.\ldots.f_{r_f}$ and $n.g_1.\ldots.g_{r_g}$ are aliases in L_R implicitly.)

In Figure 15, r/B should be context-sensitive, since $\mathsf{r}^+/\mathsf{B}^+.cs$ and $\mathsf{r}^-/\mathsf{B}^-.cs$ = true will all be tainted:

$$\ldots\ \mathsf{A1}^- \xrightarrow[\widehat{\mathsf{A1}}]{\overline{\texttt{hstore[r]}}} \mathsf{r}^- \xrightarrow{\overline{\texttt{new}}} \mathsf{B}^- \xrightarrow[\widehat{\mathsf{B}}]{\texttt{hload[f]}} \mathsf{f}^+ \xrightarrow[\check{\mathsf{B}}]{\texttt{hstore[f]}} \mathsf{B}^+ \xrightarrow{\texttt{new}} \mathsf{r}^+ \xrightarrow[\check{\mathsf{A1}}]{\texttt{hstore[r]}} \mathsf{A1}^+ \ldots \quad . \tag{20}$$

Given this path, there must be an object O^+ (not shown) flowing into $\mathsf{r}^+/\mathsf{B}^+$ in `create()` inter-procedurally and whatever $\mathsf{r}^-/\mathsf{B}^-$ points to must also be pointed to by a variable w^- outside `create()` (not shown) inter-procedurally. Both value flows are inter-procedural, as they both cross the context [$A1$] of `create()`.

- Second, we match all balanced parentheses in L_C by exploiting the regularity of L_R in $L_{RC} = L_R \cap L_C$ without having to use a context stack. By construction, we always enter a context O labeled by the opening parenthesis $\widehat{O}$ from node O^- and leave the same context O labeled by the matching closed parenthesis $\check{O}$ from node O^+ (if this can happen in the program). When solving the reachability problem in $L_{RC} = L_R \cap L_C$, whenever a balanced L_{RC}-subpath (i.e., a subpath of an L_{RC}-path with all contexts balanced out) from O^- to O^+ is found, a `match` edge (i.e., a *shortcut* from O^- to O^+, denoted $O^- \xrightarrow{\texttt{match}} O^+$) is added to $G_{\text{R-pag}}$, resulting in $G^{\texttt{match}}_{\text{R-pag}}$ (i.e., $G_{\text{R-pag}}$ augmented with these `match` edges). Thus, we also modify L_R in Figure 14(a) into $L^{\texttt{match}}_R$ as shown in Figure 14(b) by adding a new `match` transition to handle these `match` edges. However, L_C remains unchanged since a `match` edges does not have any below-edge label.
  The following lemma states that exploiting the `match` edges for solving $L^{\texttt{match}}_{RC} = L^{\texttt{match}}_R \cap L_C$ on $G^{\texttt{match}}_{\text{R-pag}}$ will still guarantee that $L^{\texttt{match}}_{RC} = L_{RC}$, since L_R is regular. (Note that its correctness is guaranteed when an object serves as both a base object of a load/store and a receiver object of a method call.)

LEMMA 3.6 [MATCH]. *Let $L^{\texttt{match}}_R$ in Figure 14(b) be obtained from L_R in Figure 14(a) as given with a* `match` *transition added. Let $G^{\texttt{match}}_{R\text{-}pag}$ be obtained from $G_{R\text{-}pag}$ (on which $L_{RC} = L_R \cap L_C$ is solved) with all* `match` *edges added. Let $L^{\texttt{match}}_{RC} = L^{\texttt{match}}_R \cap L_C$ be solved on $G^{\texttt{match}}_{R\text{-}pag}$. Then $L^{\texttt{match}}_{RC} = L_{RC}$.*

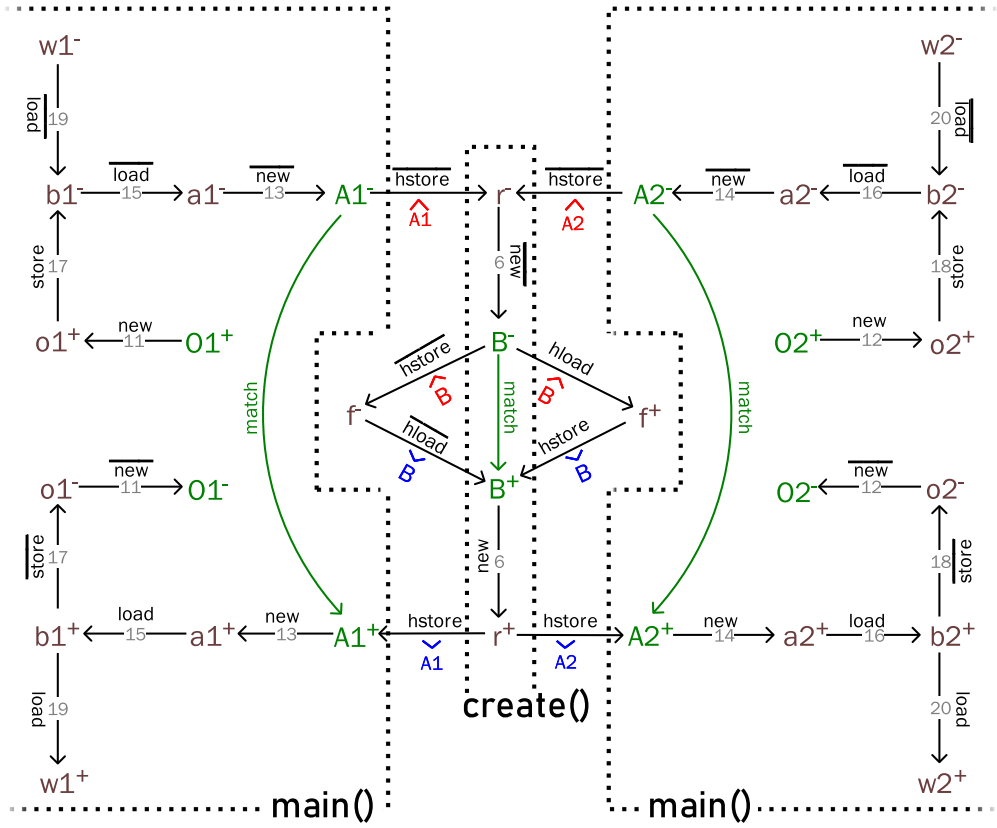


Fig. 16. The regularized PAG duplicated from Figure 15 with all match edges added according to the DFA in Figure 14(b).

Proof. By construction, $L_{RC}^{\texttt{match}} \supseteq L_{RC}$. To show $L_{RC}^{\texttt{match}} \subseteq L_{RC}$, let $P_{\texttt{match}}$ be an $L_{RC}^{\texttt{match}}$-path (i.e., a context-sensitive *flowsto* path in $L_{RC}^{\texttt{match}}$) Let P be obtained from $P_{\texttt{match}}$ by replacing every match edge $O^- \xrightarrow{\texttt{match}} O^+$ in $P_{\texttt{match}}$ with the balanced L_{RC}-subpath found when solving $L_{RC} = L_R \cap L_C$ on $G_{\text{R-pag}}$, thereby triggering this match edge to be added into $G_{\text{R-pag}}^{\texttt{match}}$. As L_R is regular, P must be an L_{RC}-path—that is, it must represent a context-sensitive *flowsto* path in L_{RC}. This proves that $L_{RC}^{\texttt{match}} \subseteq L_{RC}$. Thus, $L_{RC}^{\texttt{match}} = L_{RC}$. □

Figure 16 augments the regularized PAG shown in Figure 15 with the three match edges added during our pre-analysis. The $\texttt{B}^- \xrightarrow{\texttt{match}} \texttt{B}^+$ edge is added due to one of the following two paths traversed by Eagle,

$$\begin{aligned} &\texttt{B}^- \xrightarrow[\widehat{\texttt{B}}]{\texttt{hload[f]}} \texttt{f}^+ \xrightarrow[\check{\texttt{B}}]{\texttt{hstore[f]}} \texttt{B}^+ \\ &\texttt{B}^- \xrightarrow[\widehat{\texttt{B}}]{\overline{\texttt{hstore[f]}}} \texttt{f}^- \xrightarrow[\check{\texttt{B}}]{\overline{\texttt{hload[f]}}} \texttt{B}^+ \end{aligned}, \tag{21}$$

which exist in $G_{\text{R-pag}}$ shown in Figure 16 or due to simply the following path in G_{pag} shown in Figure 15:

$$\mathsf{B} \xrightarrow[\widehat{\mathsf{B}}]{\mathsf{hload[f]}} \mathsf{f} \xrightarrow[\widecheck{\mathsf{B}}]{\mathsf{hstore[f]}} \mathsf{B}. \tag{22}$$

At the start of Eagle (as an optimization and also to simplify the presentation of our pre-analysis), we add the match edges required for all base objects, such as B, of the store and loads that exist in the program. For every such base object O in G_{pag}, we add $O^- \xrightarrow{\text{match}} O^+$ to G_{pag} if (22) (with B substituted for O) holds.

For this example, $\mathsf{A1}^- \xrightarrow{\text{match}} \mathsf{A1}^+$ is added when (20) is traversed and $\mathsf{A2}^- \xrightarrow{\text{match}} \mathsf{A2}^+$ is added due to

$$\mathsf{A2}^- \xrightarrow[\widehat{\mathsf{A2}}]{\overline{\mathsf{hstore[r]}}} \mathsf{r}^- \xrightarrow{\overline{\mathsf{new}}} \mathsf{B}^- \xrightarrow[\widehat{\mathsf{B}}]{\mathsf{hload[f]}} \mathsf{f}^+ \xrightarrow[\widecheck{\mathsf{B}}]{\mathsf{hstore[f]}} \mathsf{B}^+ \xrightarrow{\mathsf{new}} \mathsf{r}^+ \xrightarrow[\widecheck{\mathsf{A2}}]{\mathsf{hstore[r]}} \mathsf{A2}^+ \ldots \quad . \tag{23}$$

The two paths given in (20) and earlier are balanced L_{RC}-subpaths discovered when `create()` is analyzed under its two different receivers, A1 and A2 (i.e., two different contexts during the pre-analysis).

- Third, for any two receiver objects, O_1 and O_2 of a common method, the balanced L_{RC}-subpaths corresponding to $O_1^- \xrightarrow{\text{match}} O_1^+$ and $O_2^- \xrightarrow{\text{match}} O_2^+$ are identical (except for their name differences). Observe the two balanced L_{RC}-subpaths given in (20) and (23) for causing $\mathsf{A1}^- \xrightarrow{\text{match}} \mathsf{A1}^+$ and $\mathsf{A2}^- \xrightarrow{\text{match}} \mathsf{A2}^+$ to be added, respectively.
  Therefore, the match edges for all receiver objects of a method can be added at once (when one is added).

Lemma 3.7 [Match]. *Let O_1 and O_2 be two arbitrary but fixed receiver objects of a common method. When solving $L_{RC} = L_R \cap L_C$ using* match *edges, we must have* $O_1^- \xrightarrow{\text{match}} O_1^+ \iff O_2^- \xrightarrow{\text{match}} O_2^+$.

Proof. To add $O_1^- \xrightarrow{\text{match}} O_1^+$ for a method m, Eagle for solving $L_{RC} = L_R \cap L_C$ must have found a balanced L_{RC}-subpath starting from O_1^- and ending at O_1^+, denoted $P(O_1^-, O_1^+)$. By definition, no (below-label) context in $P(O_1^-, O_1^+)$ can be O_1, as entering O_1 via O_1^- requires Eagle to leave O_1 via O_1^+, implying that $P(O_1^-, O_1^+)$ is also independent of O_1's type. As L_{RC} is object-sensitive, replacing every occurrence of O_1 in $P(O_1^-, O_1^+)$ by O_2, such that O_1 and O_2 are two different receiver objects of the same method, m, under the given hypothesis, yields a balanced L_{RC}-subpath starting from O_2^- and ending at O_2^+. So $O_2^- \xrightarrow{\text{match}} O_2^+$ is also added. □

Figure 17 gives the three rules for performing our precision-preserving pre-analysis in Algorithm 1 for supporting partial context sensitivity. [EntryCTX] and [ExitCTX] handle inter-context edges, whereas [PropCTX] handles intra-context edges (including match edges). As discussed earlier, a variable/allocation site n is selected to be context-sensitive if Eagle-P1 : $n^+.cs$ = true $\wedge$ Eagle-P2 : $n^-.cs$ = true. Built on our CFL-reachability foundation, our pre-analysis is simple conceptually as a taint analysis and implementation-wise (with several hundreds of lines of code). Thus, all rules are given with the understanding that any node that has been tainted will no longer be tainted again.

$$\frac{n \xrightarrow[\widehat{O}]{} n' \in G^{\mathsf{match}}_{\text{R-pag}}}{n'.cs = \mathsf{true}}\ [\textsc{EntryCTX}] \qquad \frac{n \xrightarrow[\check{O}]{} n' \in G^{\mathsf{match}}_{\text{R-pag}} \qquad n.cs = \mathsf{true}}{O^- \xrightarrow[\epsilon]{\mathsf{match}} O^+ \in G^{\mathsf{match}}_{\text{R-pag}}}\ [\textsc{ExitCTX}]$$

$$\frac{n \xrightarrow[\epsilon]{} n' \in G^{\mathsf{match}}_{\text{R-pag}} \qquad n.cs = \mathsf{true}}{n'.cs = \mathsf{true}}\ [\textsc{Prop}]$$

Fig. 17. Rules for performing CFL-reachability in $G^{\mathsf{match}}_{\text{R-pag}}$ (with the match edges added) according to $L^{\mathsf{match}}_{RC} = L^{\mathsf{match}}_{R} \cap L_C$.

ALGORITHM 1: Precision-Preserving Pre-Analysis for Enabling Partial Context-Sensitivity

Procedure Pre-Analysis
begin
1. Step 1: Regularize L_O into L_R as shown in Figure 14(a);
2. Step 2: Build G_{pag} for a program using Andersen's analysis, then $G_{\text{R-pag}}$ (its regularized version), and finallylet $G^{\mathsf{match}}_{\text{R-pag}}$ be obtained from $G_{\text{R-pag}}$ added with the match edges during the pre-analysis;
3. Step 3: Perform a taint analysis by solving$L^{\mathsf{match}}_{RC} = L^{\mathsf{match}}_{R} \cap L_C$ on $G^{\mathsf{match}}_{\text{R-pag}}$, where L^{match}_{R} is modified from L_R by adding a match transition as shown in Figure 14(b), according to therules in Figure 17;
4. Step 4: Select a variable/allocation site $n$in G_{pag} to be analyzedcontext-sensitively if
5. $\textsc{Eagle}\text{-P1} \wedge \textsc{Eagle}\text{-P2} : n^+.cs \wedge n^-.cs = \mathsf{true}$

[EntryCTX] handles an inter-context edge with an entry context. According to L_R in Figure 14, $n \xrightarrow[\widehat{O}]{} n'$ must be an $\overline{\mathsf{hstore}}$ edge $O^- \xrightarrow[\widehat{O}]{\overline{\mathsf{hstore}}} v^-$ or an hload edge $O^- \xrightarrow[\widehat{O}]{\mathsf{hload}} v^+$. As we must always enter a new context O from node O^- via one of these two types of edges, its target n' is tainted (indicating that it is now context-sensitive).

[ExitCTX] handles an inter-context edge with an exit context. According to L_R in Figure 14, $n \xrightarrow[\check{O}]{} n'$ must be an $\overline{\mathsf{hstore}}$ edge $v^+ \xrightarrow[\check{O}]{\mathsf{hstore}} O^+$ or an $\overline{\mathsf{hload}}$ edge $v^- \xrightarrow[\check{O}]{\overline{\mathsf{hload}}} O^+$. If n is tainted (implying that we have previously entered the underlying method, say, m, under a receiver context O^-_{prev}), then we add a match edge $O^- \xrightarrow{\mathsf{match}} O^+$ for every receiver O of m so that both Lemmas 3.6 and 3.7 are taken care of simultaneously. By Lemma 3.6, a *match* edge $O^-_{prev} \xrightarrow{\mathsf{match}} O^+_{prev}$ is added (if $O = O_{prev}$), representing that a balanced L_{RC}-subpath starting from O^-_{prev} and ending at O^+_{prev} has been found. By Lemma 3.7, a match edge $O^- \xrightarrow{\mathsf{match}} O^+$ is added for every other receiver O of m.

[Prop] handles simply the taint propagation across all intra-context edges in the standard manner.

The worst-time complexity of Eagle for a program is $O(|E^{\mathsf{match}}_{\text{R-pag}}|)$, where $G^{\mathsf{match}}_{\text{R-pag}} = (N^{\mathsf{match}}_{\text{R-pag}}, E^{\mathsf{match}}_{\text{R-pag}})$, since each method is tainted only once even in the presence of recursion. For the set of 12 Java programs evaluated in Section 4, $E^{\mathsf{match}}_{\text{R-pag}}$ is about 4.6× as big as $N^{\mathsf{match}}_{\text{R-pag}}$, on average.

Let us apply Eagle to the program in Figure 4, with its G_{pag} depicted in Figure 10 and its $G_{\text{R-pag}}$ in Figure 15. As discussed earlier, $G_{\text{R-pag}}^{\texttt{match}}$ will start as $G_{\text{R-pag}}$ initially augmented with the `match` edges for the base objects of all loads/stores (i.e., $\texttt{B}^- \xrightarrow{\texttt{match}} \texttt{B}^+$ in this example), as shown in Figure 16 (but without the other two `match` edges, $\texttt{A1}^- \xrightarrow{\texttt{match}} \texttt{A1}^+$ and $\texttt{A2}^- \xrightarrow{\texttt{match}} \texttt{A2}^+$, being added yet). We can now apply [EntryCTX] to $\texttt{A1}^- \xrightarrow[\widehat{\texttt{A1}}]{\overline{\texttt{hstore[r]}}} \texttt{r}^-$, $\texttt{A2}^- \xrightarrow[\widehat{\texttt{A2}}]{\overline{\texttt{hstore[r]}}} \texttt{r}^-$, $\texttt{B}^- \xrightarrow[\widehat{\texttt{B}}]{\overline{\texttt{hstore[f]}}} \texttt{f}^-$, and $\texttt{B}^- \xrightarrow[\widehat{\texttt{B}}]{\texttt{hload[f]}} \texttt{f}^+$ in any order. Let us try $\texttt{A1}^- \xrightarrow[\widehat{\texttt{A1}}]{\overline{\texttt{hstore[r]}}} \texttt{r}^-$ first. Immediately, $\texttt{r}^-$ is tainted (with $\texttt{r}^-.cs = \text{true}$). Then, traversing the following path (i.e., the path given in (20) via $\texttt{B}^- \xrightarrow{\texttt{match}} \texttt{B}^+$) up to $\texttt{r}^+$ by applying [Prop] in $L_R^{\texttt{match}}$, we find that $\texttt{B}^-$, $\texttt{B}^+$ and $\texttt{r}^+$ are all tainted (with $\texttt{B}^-.cs = \texttt{B}^+.cs = \texttt{r}^+.cs = \text{true}$):

$$\ldots\ \texttt{A1}^- \xrightarrow[\widehat{\texttt{A1}}]{\overline{\texttt{hstore[r]}}} \texttt{r}^- \xrightarrow{\overline{\texttt{new}}} \texttt{B}^- \xrightarrow{\texttt{match}} \texttt{B}^+ \xrightarrow{\texttt{new}} \texttt{r}^+ \xrightarrow[\widecheck{\texttt{A1}}]{\texttt{hstore[r]}} \texttt{A1}^+ \ldots \quad . \tag{24}$$

We can now apply [ExitCTX] to the last edge in the path above. As $\texttt{r}^+.cs = \text{true}$, we introduce a `match` edge, $\texttt{A1}^- \xrightarrow{\texttt{match}} \texttt{A1}^+$. Due to also the existence of $\texttt{r}^+ \xrightarrow[\widecheck{\texttt{A2}}]{\texttt{hstore[r]}} \texttt{A2}^+$, $\texttt{A2}^- \xrightarrow{\texttt{match}} \texttt{A2}^+$ is added at the same time. As $\texttt{A1}^-.cs = \texttt{A2}^-.cs = \text{false}$, [Prop] is not applicable to $\texttt{A1}^- \xrightarrow{\texttt{match}} \texttt{A1}^+$ and $\texttt{A2}^- \xrightarrow{\texttt{match}} \texttt{A2}^+$. Thus, no more taint propagation will be needed beyond $\texttt{A1}^-$ and $\texttt{A2}^-$. Now, $G_{\text{R-pag}}^{\texttt{match}}$ looks like the regularized PAG shown in Figure 16, which is the final one obtained for this example. If we apply [EntryCTX] now to $\texttt{A2}^- \xrightarrow[\widehat{\texttt{A2}}]{\overline{\texttt{hstore[r]}}} \texttt{r}^-$, no more taint propagation is needed, as the same three nodes, $\texttt{B}^-$, $\texttt{B}^+$, and $\texttt{r}^+$, have already been tainted (Lemma 3.7):

$$\ldots\ \texttt{A2}^- \xrightarrow[\widehat{\texttt{A2}}]{\overline{\texttt{hstore[r]}}} \texttt{r}^- \xrightarrow{\overline{\texttt{new}}} \texttt{B}^- \xrightarrow{\texttt{match}} \texttt{B}^+ \xrightarrow{\texttt{new}} \texttt{r}^+ \xrightarrow[\widecheck{\texttt{A2}}]{\texttt{hstore[r]}} \texttt{A2}^+ \ldots \quad . \tag{25}$$

Next, applying [EntryCTX] to $\texttt{B}^- \xrightarrow[\widehat{\texttt{B}}]{\overline{\texttt{hstore[f]}}} \texttt{f}^-$ and $\texttt{B}^- \xrightarrow[\widehat{\texttt{B}}]{\texttt{hload[f]}} \texttt{f}^+$ will give rise to $\texttt{f}^-.cs = \texttt{f}^+.cs = \text{true}$. As $\texttt{B}^-.cs = \texttt{B}^+.cs = \text{true}$, no taint propagation is needed along $\texttt{B}^- \xrightarrow{\texttt{match}} \texttt{B}^+$ by [Prop]. As mentioned in Section 3.2, the fields in a program are treated uniformly as a special case of variables in its PAG. As $\texttt{f}^-.cs = \texttt{f}^+.cs = \text{true}$, f needs to be analyzed context-sensitively. In some library code, if there are just stores but never any load for a field f, in which case $\texttt{f}^+.cs = \text{true}$ but $\texttt{f}^-.cs = \text{false}$, then these stores can be modeled context-insensitively as just the stores into f (without any base object). We have decided not to emphasize this point earlier, since its impact on performance in real code is small.

We have just illustrated our pre-analysis with an example. When moving from $L_{FC} = L_F \cap L_C$ to $L_{RC} = L_R \cap L_C$, we have regularized L_F into L_R over-approximately but strived to enforce L_C fully without k-limiting. This is critical to obtaining a pre-analysis that is both effective and efficient, as motivated earlier and evaluated later in the article.

The following theorem, which is an analogue of Theorem 3.4, states that Eagle is precision-preserving. As Eagle operates on a pre-built PAG for a program without actually computing its points-to information, all of its methods are analyzed (since they are all assumed to be reachable conservatively according to the rules in Figure 17).

Theorem 3.8 [Precision for Eagle]. *Let Eagle be the pre-analysis for solving $L_{RC}^{\texttt{match}} = L_R^{\texttt{match}} \cap L_C$ (without k-limiting) on $G_{R\text{-}pag}^{\texttt{match}}$ according to Algorithm 1. For a program P, a variable/allocation site in its method m is selected to be context-sensitive if Eagle-P1 : $n^+.cs = true \wedge$ Eagle-P2 : $n^-.cs = true$.*

Let k-obj$_{RC}$ be the resulting version of k-obj. *Then k-obj$_{EAGLE}$ has exactly the same precision as k*-obj: $\overline{pt}_{k\text{-}obj_{EAGLE}}(v) = \overline{pt}_{k\text{-}obj}(v)$ *for every variable v in P.*

Proof. Let CI_{EAGLE} (CI_{OC}) be the set of variables/allocation sites in P that are selected to be context-insensitive by Eagle (OC_{pre}). By Lemma 3.5, $L_R \supseteq L_O$, implying that $L_R \cap L_C = L_{RC} \supseteq L_{OC} = L_O \cap L_C$. By Lemma 3.6, $L_{RC}^{\texttt{match}} = L_{RC}$, where L_{RC} is solved on $G_{\text{R-pag}}$. If $n^+.cs \wedge n^-.cs \neq \text{true}$ according to Algorithm 1, then there cannot be a *flowsto* path, O *flowsto* w, that passes through n in L_R—that is, $O^+ \xrightarrow{L_R} n^+ \wedge w^- \xrightarrow{L_R} n^-$ in L_R such that both $O^+ \xrightarrow{L_R} n^+$ holds inter-procedurally (Eagle-P1) and $w^- \xrightarrow{L_R} n^-$ holds inter-procedurally (Eagle-P2). As $L_{RC} \supseteq L_{OC}$ (i.e., Eagle is more conservative than OC_{pre}), then $CI_{\text{EAGLE}} \subseteq CI_{OC}$. Finally, a further application of Theorem 3.4 concludes the proof. □

4 EVALUATION

As Eagle represents the first provably precision-preserving pre-analysis for accelerating *k-obj*, our evaluation focuses mainly on understanding and analyzing the efficiency gains achieved by Eagle in scaling *k-obj* for large Java programs, providing benefits for three representative client analyses: call graph construction, may-fail-casting, and polymorphic call detection. As a side effect, our evaluation also serves to validate the precision guarantees provided by Eagle.

4.1 Experimental Settings

We have implemented Eagle and all related analyses used in our evaluation in Soot [43], a program analysis and optimization framework for Java and Android programs, on top of its context-insensitive Andersen's pointer analysis, Spark [15], and its object-sensitive version, Obj-Sens-Soot, initially developed by MIT [8]. Eagle, which relies on Spark to pre-build G_{pag} for a program, is conceptually simple and easily implementable, with just about 700 lines of Java code for its core algorithm (Algorithm 1). The baseline, *k-obj*, is Obj-Sens-Soot, which is modified simply to support partial context sensitivity required by Eagle. The configuration used for running Spark is available in the open source release of our Eagle pointer analysis framework [7]. This configuration is also used by our Eagle pre-analysis and *k-obj* (as both the baseline and the modified version for supporting partial context sensitivity prescribed by Eagle).

Eagle represents a pre-analysis for improving the efficiency of *k-obj* by identifying the variables/allocation sites in a program that can be analyzed context-insensitively without incurring any precision loss. In this work, we will therefore not apply any traditional heuristics or experience-based techniques beforehand to pre-select certain variables/allocation sites in the program as context-insensitive (e.g., by merging certain objects by their dynamic types and then marking the merged ones as context-insensitive). In practice, such heuristics may often represent some good engineering choices, but they are not precision-preserving (even through they often make *k-obj* run faster). In a way, this research can be understood as contributing an approach for identifying systematically which variables/allocation sites in a program can be analyzed by *k-obj* context-insensitively while still enabling *k-obj* to maintain its precision.

4.1.1 Static Fields and Methods. We have formalized Eagle for a simple Java-like language in Figure 6, where only instance fields and methods are supported. For Java, we must decide how static fields and methods are analyzed.

In *k-obj*, static fields are always analyzed context-insensitively as they are global variables (accessible in all possible contexts). Thus, Eagle assumes all static fields to be handled context-insensitively by *k-obj*. During its pre-analysis (Algorithm 1), for every node n^s in $G_{\text{R-pag}}$, where $s \in \{+, -\}$, such that n is a static field, $n^+.cs$ and $n^-.cs$ are permanently set to false. During

the main analysis with *k-obj*, such nodes will always be handled context-insensitively. Hence, the Eagle-guided *k-obj* pointer analyses will still be precision-preserving (as Theorem 3.8 still holds).

As for static methods, we assume that *k-obj* adopts a recent approach [8, 35, 40] for enabling their context-sensitive analysis. According to this approach, a static method *m* is analyzed by using the contexts of *m*'s closest callers that are instance methods on the call stack. To support this abstraction, we can extend straightforwardly the simple Java-like language given in Figure 6 as follows. Let `Root_Class` be a pseudo class, which is constructed to be a superclass of all other classes in the program, such that `Root_Class` contains all static methods in the program. Let us postulate further that all of these static methods are analyzed by Eagle as if they were instance methods. Specifically, when analyzing a call to a static method $m()$, which is interpreted as $this.m()$, the *this* variable in the call will represent naturally the receivers of *m*'s closest callers that are instance methods (on the call stack). Then the Eagle-guided *k-obj* pointer analysis will still be precision-preserving (as Theorem 3.8 still holds). Previously [23], we evaluated the performance benefits of Eagle in accelerating *k-obj* in which all static methods are analyzed context-insensitively [27].

4.1.2 Dynamic Language Features. For a simple Java-like language in Figure 6, all of our pointer analysis formulations are sound. For Java, however, its Java reflection and Java Native Interface (JNI) are two major obstacles for achieving soundness, in practice [21]. On one hand, modeling these two dynamic language features inadequately can render much of the codebase invisible for analysis. On the other hand, modeling both features conservatively can render the pointer analysis unscalable. In our open source implementation [7], we have handled both features similarly as in the prior work [17, 29, 30, 40]. For reflection, static reflection resolution is sometimes done during a pointer analysis to improve its soundness. However, this can introduce significant overheads due to the time and space needed in analyzing potentially a large number of spurious reflective calls identified conservatively, making it hard to compare the effectiveness of different pointer analyses in analyzing real-world applications [19]. Thus, we have decided to resolve Java reflection by using TamiFlex, a dynamic reflection analysis tool, [4], as is often done in the pointer analysis literature [17, 29, 30, 40]. This covers `ClassForName`, `ClassNewInstance`, `ConstructorNewInstance`, `MethodInvoke`, `FieldSet`, and `FieldGet`, a set of the mostly commonly used methods provided in the Java reflection API. To model native code, we use their method summaries [7] except for the reflective methods already handled by TamiFlex.

For a given program, all pointer analyses compared will operate on exactly the same source code obtained after reflection resolution and native code modeling. How to handle these two dynamic language features is orthogonal.

4.1.3 Constraint Resolution. In Soot [15], both Spark and Obj-Sens-Soot rely on an incremental worklist solver to resolve the points-to constraints (also known as *pointer assignment constraints*) during the pointer analysis. To enable partial context sensitivity supported by Eagle, we have modified this solver so that we can handle differently the variable/allocation sites residing in a method, depending on whether they are analyzed context-sensitively or not.

Specifically, Obj-Sens-Soot is modified to support partial context sensitivity as follows. Let v be a variable contained in a method *m*. If v is context-sensitive (according to Eagle), *ek-obj* will proceed exactly as *k-obj* with v identified as (c, v), where c is a context under which *m* is analyzed. If v is context-insensitive (according to Eagle), then v is simply represented as $([\,], v)$. Objects are handled context-sensitively and context-insensitively in a similar manner.

Table 3. Benchmark Statistics

Program	Total		Reachable Code Obtained by Spark (i.e., Andersen's Analysis)						
	#Classes	#Methods	#Classes	#Methods	#Call Edges	#Variables	#Allocs	#Loads	#Stores
antlr	5464	53089	1160	8019	56722	53178	9066	11774	5677
bloat	2552	25172	1321	9287	66890	57422	9154	11528	6107
chart	6973	72771	2110	14000	75259	79346	14851	15669	15844
eclipse	4882	50317	2683	22867	185507	161027	21230	44375	16413
fop	7795	66990	1583	7839	39778	40537	7921	7414	5629
luindex	2730	24961	1148	7231	38957	39229	7267	8206	5236
lusearch	2729	24957	1093	6803	36613	36601	6904	7293	4995
pmd	6807	63656	1816	12218	69311	74103	10746	20229	9375
xalan	6476	62732	1396	8363	44196	44493	8377	8646	6138
checkstyle	8305	77658	1828	12597	79903	82818	12601	22464	10021
findbugs	7856	73104	2177	13959	83314	89495	13432	22048	10836
JPC	5861	59073	2077	13856	71826	73299	12162	16026	10605

4.1.4 Benchmarking. We have used a set of 12 Java programs, including nine benchmarks from DaCapo2006 [3], and three Java applications, `checkstyle`, `findbugs`, and `JPC`, which are commonly used in evaluating *k-obj* [11, 12, 36, 39, 40]. For the **Java Runtime Environment (JRE)**, we have used `JRE1.6.0_45` to analyze all 12 programs (since the DaCapo2006 benchmarks rely on only an older version of JRE). We consider three representative client analyses: call graph construction, may-fail-casting, and polymorphic call detection.

Table 3 presents the basic information regarding the 12 benchmark programs (with the JRE library considered as part of each program). In the first two columns, we give the total number of classes and methods in each program. In the last seven columns, we characterize the code discovered for each benchmark by Spark (i.e., the analysis of Andersen [1]), including the number of reachable classes, methods, call edges, variables, allocation sites, loads, and stores. In particular, the number of call edges (#Call Edges) discovered by Andersen's analysis in a program represents roughly the size of the code to be analyzed by all different object-sensitive pointer analyses compared and evaluated in this article.

For *k-obj*, we consider three versions with different degrees of context sensitivity with k ranging over $\{1, 2, 3\}$. As *k-obj* is typically unscalable beyond $k = 2$, *2-obj* is often used but *3-obj* is rarely considered [17, 29, 30, 40].

We measure the precision of a context-sensitive pointer analysis as in other works [36, 37, 40] in terms of four metrics, which can all be regarded as being obtained in terms of the context-insensitive points-to information computed according to (3). There are two intrinsic metrics: #Call Edges (the number of call graph edges discovered for call graph construction) and #Avg Points-to Size (the average size of points-to sets computed by (3) with the contexts dropped). There are also two end-user visible metrics: #Poly Calls (the number of polymorphic calls discovered for polymorphic call detection) and #May Fail Casts (the number of type casts that may fail for may-fail-casting). Note that all of these metrics are presented for both the application code and the library code included in a program, as has been done in many recent pointer analysis works for Java programs [11, 12, 17, 36, 39, 40].

We measure the efficiency of a pointer analysis in terms of the analysis time elapsed in analyzing a program to completion (as an average of three runs). The time budget set for analyzing a program is 24 hours.

We have carried out all of our experiments on a Xeon E5-1660 3.2GHz machine with 256 GB of RAM.

4.2 Results and Analysis

We evaluate Eagle on accelerating *k-obj*. Table 4 presents the main results for the two main analyses, where, for each $k \in \{1, 2, 3\}$, *k-obj* is the baseline and *ek-obj* is the version of *k-obj* guided by Eagle to improve its performance by supporting partial context sensitivity. Table 5 then presents the results for our pre-analysis.

4.2.1 Pre-Analysis. Table 5 gives the results. Eagle performs its pre-analysis by using the points-to information pre-computed by `Spark` [15], implemented as a context-insensitive Andersen's analysis. According to Columns 2 and 3, `Spark`'s analysis times are negligible relative to *k-obj*'s analysis times across the 12 programs. For Eagle, its pre-analysis times exclude the times for running `Spark`. Compared with `Spark`, Eagle is always faster (as it is linear in terms of the number of pointer assignment edges in the program). We have also collected the maximum amount of memory consumed by Eagle (using the JVM API) for processing each program during its pre-analysis.

4.2.2 Precision. By Theorem 3.8, Eagle is precision-preserving as it reasons about CFL-reachability to support partial context sensitivity. For every program that is scalable under *k-obj*, *ek-obj* also produces scalably the same context-insensitive points-to information, as Eagle guarantees that $\overline{pt}_{ek\text{-}obj}(v) = \overline{pt}_{k\text{-}obj}(v)$ for every variable v in the program, and consequently the same results for all four precision-related metrics considered.

4.2.3 Efficiency. Let us now understand and analyze the speedups achieved by Eagle. In Table 4, the speedup of *ek-obj* for a program is given with *k-obj* as the baseline. The analysis time spent by *ek-obj* on analyzing a program does not include the times taken by `Spark` and Eagle (Table 5). There are two reasons. First, the times taken by `Spark` and Eagle are relatively small for large programs. Second, these times are amortized in practice. The context-insensitive points-to information produced by `Spark` can be reused by a large number of client analyses (e.g., those that make use of alias and def-use information). Similarly, the pre-analysis results obtained by Eagle can be used to accelerate many client analyses that employ *k-obj* as its underlying pointer analysis for all different values of *k*.

According to Columns 3 through 8, *ek-obj* runs significantly faster than *k-obj* for all 12 programs under all configurations (for which *ek-obj* is scalable). For each $k \in \{1, 2, 3\}$, *ek-obj* can scalably analyze no fewer programs than *k-obj*: 12 in both cases ($k = 1$), 11 under *k-obj* but 12 under *ek-obj* ($k = 2$), and 5 under *k-obj* but 9 under *ek-obj* ($k = 3$). For the programs that can be scalably analyzed by *k-obj*, totaling 12 ($k = 1$), 11 ($k = 2$), and 5 ($k = 3$), *ek-obj* exhibits increasingly better scalability, yielding the speedups of 3.8× ($k = 1$), 5.2× ($k = 2$), and 6.6× ($k = 3$), on average. For the four programs that can be analyzed scalably by *ek-obj* but not by *k-obj* under our pre-set 24-hour time budget ($k = 3$), *ek-obj* has succeeded in analyzing all of them in just under 9.5 hours in total.

In the following, let us analyze the reasons behind the speedups of *ek-obj* over *k-obj* even without any loss of precision.

Figure 18 depicts the percentage distribution of methods with different kinds of context sensitivity in a program selected by Eagle. Eagle has successfully identified a large percentage of partially context-sensitive methods in all of the programs, ranging from 33% (in `fop`) to 42% (in `eclipse`), with an average of 37%. These are the methods that would be analyzed either context-sensitively or context-insensitively in its entirety previously [9, 12, 17, 36], resulting in potentially a loss of either efficiency or precision. Figure 19 depicts the percentage distribution of variables/allocation sites in a program that are selected to be context-sensitive and

Table 4. Analysis Results for Accelerating *k-obj* with Eagle

Program	Metrics	1-obj	e1-obj	2-obj	e2-obj	3-obj	e3-obj
antlr	Time(s)	26.9	8.8	129.9	24.4	2015.5	238.6
	Speedup	1.0×	3.1×	1.0×	5.3×	1.0×	8.4×
	#Call Edges	54342	54342	50145	50145	50113	50113
	#Poly Calls	1866	1866	1636	1636	1627	1627
	#May Fail Casts	928	928	484	484	428	428
	Avg Points-to size	27.50	27.50	6.17	6.17	5.12	5.12
bloat	Time(s)	28.1	13.0	711.9	351.4	-	7727.4
	Speedup	1.0×	2.2×	1.0×	2.0×	-	-
	#Call Edges	63277	63277	55455	55455	-	55221
	#Poly Calls	1998	1998	1562	1562	-	1548
	#May Fail Casts	1869	1869	1262	1262	-	1169
	Avg Points-to size	49.06	49.06	14.71	14.71	-	13.96
chart	Time(s)	157.6	35.8	619.0	102.3	-	-
	Speedup	1.0×	4.4×	1.0×	6.1×	-	-
	#Call Edges	71268	71268	62766	62766	-	-
	#Poly Calls	2303	2303	1933	1933	-	-
	#May Fail Casts	1965	1965	1150	1150	-	-
	Avg Points-to size	89.86	89.86	4.86	4.86	-	-
eclipse	Time(s)	522.1	132.3	-	16283.3	-	-
	Speedup	1.0×	3.9×	-	-	-	-
	#Call Edges	176685	176685	-	161049	-	-
	#Poly Calls	10259	10259	-	9653	-	-
	#May Fail Casts	4614	4614	-	3574	-	-
	Avg Points-to size	70.36	70.36	-	19.85	-	-
fop	Time(s)	18.1	4.2	75.8	13.5	1275.9	195.1
	Speedup	1.0×	4.3×	1.0×	5.6×	1.0×	6.5×
	#Call Edges	37389	37389	33319	33319	33287	33287
	#Poly Calls	1087	1087	842	842	833	833
	#May Fail Casts	723	723	389	389	341	341
	Avg Points-to size	19.03	19.03	3.73	3.73	3.63	3.63
luindex	Time(s)	13.2	3.7	60.8	14.1	1052.1	178.6
	Speedup	1.0×	3.6×	1.0×	4.3×	1.0×	5.9×
	#Call Edges	36591	36591	32496	32496	32464	32464
	#Poly Calls	1149	1149	916	916	907	907
	#May Fail Casts	723	723	367	367	317	317
	Avg Points-to size	14.67	14.67	3.72	3.72	3.62	3.62
lusearch	Time(s)	12.0	3.6	57.2	10.7	974.1	178.5
	Speedup	1.0×	3.3×	1.0×	5.3×	1.0×	5.5×
	#Call Edges	34288	34288	30294	30294	30262	30262
	#Poly Calls	1036	1036	807	807	798	798
	#May Fail Casts	678	678	357	357	309	309
	Avg Points-to size	14.34	14.34	3.80	3.80	3.70	3.70

(Continued)

Table 4. Continued

Program	Metrics	1-obj	e1-obj	2-obj	e2-obj	3-obj	e3-obj
pmd	Time(s)	46.6	11.1	202.9	34.2	4218.8	636.1
	Speedup	1.0×	4.2×	1.0×	5.9×	1.0×	6.6×
	#Call Edges	66044	66044	59106	59106	59036	59036
	#Poly Calls	2792	2792	2375	2375	2366	2366
	#May Fail Casts	2041	2041	1415	1415	1361	1361
	Avg Points-to size	29.74	29.74	4.98	4.98	4.81	4.81
xalan	Time(s)	22.2	8.2	554.3	142.6	-	8386.0
	Speedup	1.0×	2.7×	1.0×	3.9×	-	-
	#Call Edges	40738	40738	36377	36377	-	36345
	#Poly Calls	1254	1254	1021	1021	-	1012
	#May Fail Casts	829	829	443	443	-	395
	Avg Points-to size	19.36	19.36	4.02	4.02	-	3.93
checkstyle	Time(s)	86.7	19.4	7766.8	1572.8	-	-
	Speedup	1.0×	4.5×	1.0×	4.9×	-	-
	#Call Edges	74378	74378	65819	65819	-	-
	#Poly Calls	2559	2559	2219	2219	-	-
	#May Fail Casts	1646	1646	1117	1117	-	-
	Avg Points-to size	31.40	31.40	6.13	6.13	-	-
findbugs	Time(s)	80.6	16.2	638.0	80.1	-	16682.6
	Speedup	1.0×	5.0×	1.0×	8.0×	-	-
	#Call Edges	78142	78142	67099	67099	-	66418
	#Poly Calls	3119	3119	2624	2624	-	2582
	#May Fail Casts	2109	2109	1340	1340	-	1159
	Avg Points-to size	38.36	38.36	5.36	5.36	-	4.77
JPC	Time(s)	101.6	24.2	388.8	64.6	-	907.7
	Speedup	1.0×	4.2×	1.0×	6.0×	-	-
	#Call Edges	68196	68196	59692	59692	-	58021
	#Poly Calls	2669	2669	2232	2232	-	2091
	#May Fail Casts	1797	1797	1243	1243	-	1092
	Avg Points-to size	70.75	70.75	5.20	5.20	-	4.45

For a given $k \in \{1, 2, 3\}$, the speedups of *k-obj* and *ek-obj* (i.e., Eagle-guided *k-obj*) are normalized with *k-obj* as the baseline.

context-insensitive by Eagle. On average, the percentage of the number of context-insensitive variables/allocation sites over the total in a program is 40%. These results provide solid evidence for the performance benefits achieved by Eagle in accelerating *k-obj* with partial context sensitivity (at the granularity of variables/allocation sites) while maintaining its precision.

Table 6 gives the number of context-sensitive facts inferred in each program by *k-obj* and *ek-obj*, demonstrating the significant context-sensitive fact reductions achieved by *ek-obj* over *k-obj* across all of the programs. Although the speedup achieved by *ek-obj* over *k-obj* is not linearly proportional to the degree of context-sensitive facts reduced in a program (in theory), increasing the number of context-insensitive variables/allocation sites analyzed in the program, as motivated with the example in Figure 1, can usually accelerate *k-obj* significantly. For example, *e2-obj* has succeeded in analyzing a large number of variables/allocation sites in each program context-insensitively. As a result, for each program analyzed by *2-obj*, *e2-obj* has inferred

Table 5. Time and Memory Consumption of the Eagle Pre-Analysis

Program	Time (seconds)		Memory (MB)
	Spark	Eagle	Eagle
antlr	10	3	1539
bloat	10	3	1532
chart	17	8	1651
eclipse	54	32	3169
fop	7	2	860
luindex	7	2	862
lusearch	6	2	951
pmd	12	7	1636
xalan	8	2	818
checkstyle	15	8	1656
findbugs	15	9	1641
JPC	15	5	1758

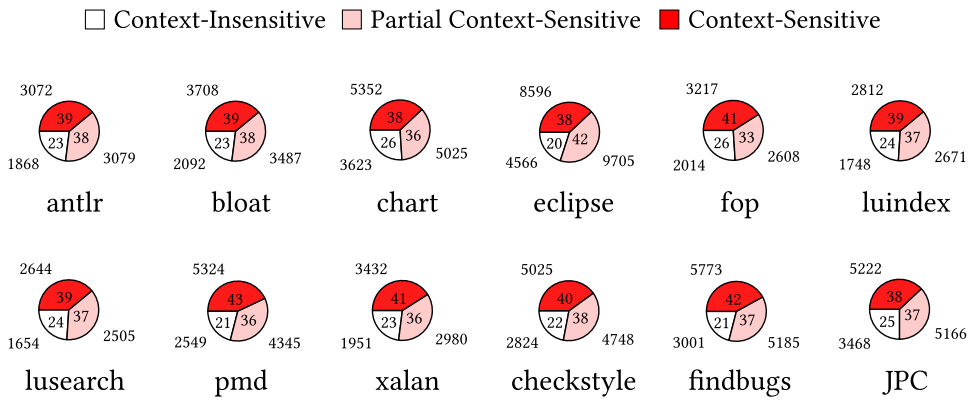


Fig. 18. Percentage distribution (and number) of methods with different kinds of context sensitivity selected by Eagle.

significantly fewer context-sensitive facts, thereby achieving a significant speedup over *2-obj*. Let us examine two programs, `findbugs` and `bloat`, analyzed by *2-obj* over *e2-obj*. For `findbugs`, if we examine the number of context-sensitive facts inferred, we have 885.0M (for 2-*obj*) and 82.7M (for *e2-obj*), with the ratio of 2-*obj* over *e2-obj* being 10.7. As a result, the speedup of *e2-obj* over 2-*obj* is 8.0×, as shown in Table 4. As for `bloat`, the ratio of 2-*obj* over *e2-obj* is 2.4 only in terms of context-sensitive facts inferred. As a result, the speedup of *e2-obj* over 2-*obj* is 2.0×, which is less pronounced . For programs such as bloat, where some tree data structures are traversed by the visitor design pattern, context sensitivity is known to be not that useful.

5 RELATED WORK

Currently, a popular approach improves the efficiency of *k-obj* by sacrificing its precision [9, 12, 17, 18, 36]. This is accomplished by exploiting different heuristics-based pre-analyses, such as machine learning [12], user-provided hints [9, 36], and pattern matching [17, 18], to determine which

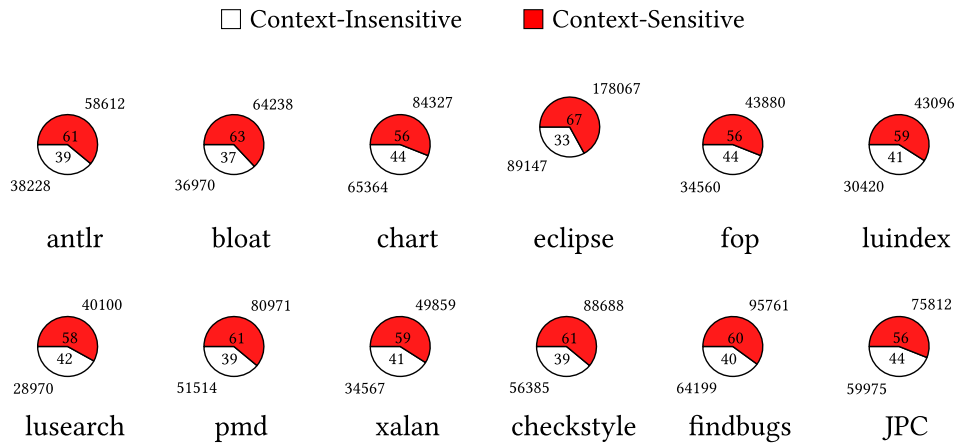


Fig. 19. Percentage distribution (and number) of variables/allocation sites with different kinds of context sensitivity selected by Eagle.

Table 6. Number of Context-Sensitive Facts for *2-obj* and *e2-obj*

Program	Metrics	*2-obj*	*e2-obj*	Program	Metrics	*2-obj*	*e2-obj*
antlr	#Var Points-to Relations	308.5M	18.8M	lusearch	#Var Points-to Relations	145.8M	8.2M
	#Field Points-to Relations	1368K	572K		#Field Points-to Relations	822K	318K
	#Call Edges	10.2M	4.1M		#Call Edges	4.1M	1.7M
	Total	320.1M	23.4M		Total	150.7M	10.3M
bloat	#Var Points-to Relations	387.9M	140.5M	pmd	#Var Points-to Relations	414.1M	34.4M
	#Field Points-to Relations	4985K	4232K		#Field Points-to Relations	3634K	1507K
	#Call Edges	41.7M	35.4M		#Call Edges	12.3M	5.5M
	Total	434.6M	180.2M		Total	430.0M	41.4M
chart	#Var Points-to Relations	1252.6M	53.7M	xalan	#Var Points-to Relations	1345.9M	128.3M
	#Field Points-to Relations	3569K	1161K		#Field Points-to Relations	8756K	3750K
	#Call Edges	39.5M	15.1M		#Call Edges	41.3M	20.1M
	Total	1295.7M	70.0M		Total	1395.9M	152.1M
eclipse	#Var Points-to Relations	-	1909.9M	checkstyle	#Var Points-to Relations	1958.0M	285.4M
	#Field Points-to Relations	-	175697K		#Field Points-to Relations	7757K	3198K
	#Call Edges	-	359.7M		#Call Edges	284.7M	113.4M
	Total	-	2445.2M		Total	2250.5M	401.9M
fop	#Var Points-to Relations	165.8M	9.1M	findbugs	#Var Points-to Relations	844.3M	65.3M
	#Field Points-to Relations	1051K	365K		#Field Points-to Relations	4669K	2899K
	#Call Edges	4.9M	2.0M		#Call Edges	36.0M	14.5M
	Total	171.8M	11.5M		Total	885.0M	82.7M
luindex	#Var Points-to Relations	157.2M	9.0M	JPC	#Var Points-to Relations	1202.7M	42.2M
	#Field Points-to Relations	873K	339K		#Field Points-to Relations	2392K	941K
	#Call Edges	4.4M	1.9M		#Call Edges	21.7M	9.8M
	Total	162.4M	11.2M		Total	1226.8M	52.9M

For a given program, #Var Points-to Relations is the number of context-sensitive points-to relations defined in (1), #Field Points-to Relations is the number of context-sensitive field points-to relations, and #Call Edges is the number of context-sensitive call edges found by each analysis in the program.

methods in a program should be analyzed by *k-obj* context-sensitively or context-insensitively in their entirety. In particular, Zipper [17, 18] is designed to accelerate *k-obj* by trading precision for efficiency. Zipper performs a pre-analysis by exploiting pattern-matching heuristics to determine if a method should be analyzed by *k-obj* context-sensitively or context-insensitively. Zipper guarantees that $\overline{pt}_{\text{Zipper}}(v) \supseteq \overline{pt}_{k\text{-}obj}(v)$ for every variable v in the program, but the = in $\supseteq$ often fails to hold. When it is often necessary for *k-obj* to analyze a method context-sensitively only for some of its variables/allocation sites, Zipper will end up analyzing it as a whole either context-sensitively (by losing efficiency) or context-insensitively (by losing precision). In this work, we have taken a different approach for developing a pre-analysis that can accelerate *k-obj* substantially while maintaining its precision. Eagle guarantees that $\overline{pt}_{\text{Eagle}}(v) = \overline{pt}_{k\text{-}obj}(v)$ for every variable v in the program. This is made possible by enabling *k-obj* to analyze a method with partial context sensitivity, based on a new CFL-reachability formulation of *k-obj*. In addition, Zipper was also used to accelerate a type-sensitive variant of *k-obj* [35], again without being able to provide any precision guarantees. In contrast, Eagle can also accelerate this particular type-sensitive variant of *k-obj* except it will achieve the same or better precision. For more details, we refer to the work of Lu [22].

In comparison with our earlier conference paper [23], we have made a number of significant changes in this article. First, we have completely reformalized the Eagle-style pre-analysis by interleaving its incremental development with a number of fully proved theorems, highlighting the technical challenges faced, and providing new insights behind its CFL-reachability formulation. Second, we have added a number of carefully designed new code examples to facilitate understanding the subtleties and complexities of our CFL-reachability-based precision-preserving approach. Third, we have demonstrated that Eagle can also achieve a significant precision-preserving acceleration of a new baseline *k-obj*, in which static methods are analyzed context-sensitively (rather than context-insensitively as in the work of Lu and Xue [23]). Fourth, we have provided additional experimental results to understand the speedups achieved by Eagle. Finally, we have open sourced our entire Eagle analysis framework [7] to enable other researchers to leverage it to develop new CFL-reachability-based pointer analyses (and other down-stream client analysis tools).

Orthogonally, some recent efforts [11, 39, 46] aim to improve the precision of *k-obj* at the expense of efficiency. In the work of Zhang et al. [46], *k-obj* is tried with increasingly larger values of k to prove that certain callsites are monomorphic. Bean [39] represents a pre-analysis that selects the calling contexts of a method by looking beyond its receiver's k-most-recent allocation sites. This idea has recently been explored further by applying machine learning techniques [11].

Mahjong [40] merges so-called type-consistent allocation sites to improve the efficiency of *k-obj* while achieving nearly the same precision for some type-dependent clients, such as call graph construction, devirtualization, and may-fail-casting. However, it achieves this at the expense of some significant precision loss for aliases. In contrast, Eagle is fully precision-preserving, ensuring that any Eagle-guided *k-obj* achieves exactly the same precision as *k-obj*.

In the work of Jeong et al. [12], machine learning techniques are applied to select the lengths of calling contexts for different methods analyzed by *k-obj* tailored for a particular client (e.g., may-fail-casting). In contrast, Eagle represents a general-purpose technique that selects the variables/allocation sites in a program to be analyzed context-sensitively by *k-obj* (for any value of k) by reasoning about the value flow in the program based on CFL-reachability. By identifying certain allocation sites to be modeled context-insensitively, as illustrated in Figure 1, Eagle will also be able to reduce the lengths of the calling contexts for method calls with the receiver objects created at these allocation sites.

In the work of Wei and Ryder [44], several context abstractions, 1-callsite, 1-object, or i-th-parameter, are applied adaptively to each function in JavaScript programs guided by a machine

learning–based pre-analysis. This approach achieves a better precision than any context-sensitive analysis that employs one of these context abstractions alone.

Oh et al. [28] also apply a pre-analysis to determine which callsites to a function in a C program need to be analyzed context-sensitively. Their basic idea is to first perform a pre-analysis to estimate the impact of context sensitivity on the main analysis's precision and then use this information gathered to find out when and where the main analysis should turn on or off its context sensitivity in the program being analyzed.

PAGs have been widely used in formulating a number of callsite-context-sensitive pointer analyses in terms of CFL-reachability [5, 33, 37, 41, 45]. In addition, such PAG-like representations have also been recently used to facilitate the incrementalization and parallelization of Andersen's pointer analysis [20].

CFL-reachability (context-free language reachability) has been widely used in developing and understanding a range of pointer analysis algorithms. Traditionally, a pointer analysis is formulated as the intersection of two CFLs, $L_F \cap L_C$, where L_F describes field accesses as balanced parentheses and L_C enforces callsite sensitivity by matching method calls and returns. Earlier, a demand-driven pointer analysis for Java was given, initially without context sensitivity [38] and subsequently extended with context sensitivity [37]. Later, a number of optimizations were proposed for improving its efficiency, by, for example, avoiding redundant path traversals [33], skipping the alias analysis for the base variables of a pair of load and store statements that are determined to be must-not-aliases (during a pre-analysis) [45], and performing the demand-driven pointer analysis incrementally [24]. FlowCFL [25] represents a new framework for reasoning about type-based reachability analysis in the presence of mutable data, but still based on callsite sensitivity. Recently, a whole-program pointer analysis has been introduced for Java [41], by taking a transformational approach rather than a context-string-based approach that is usually adopted in the pointer analysis community [26, 27, 35, 40]. Unlike these previous investigations, our major technical contributions are a new CFL formulation for object sensitivity and a lightweight CFL-reachability-based pre-analysis that can accelerate *k-obj* while preserving its precision.

Eagle can be understood conceptually as performing a kind of static taint analysis as it essentially classifies or taints the variables/objects in a program as either context-sensitive or context-insensitive. However, Eagle differs fundamentally from existing taint analysis techniques on bug detection [2, 10, 42], as it is formulated as a pre-analysis to enable *k-obj* to perform its object-sensitive pointer analysis with partial context sensitivity.

6 CONCLUSION

In this article, we have introduced Eagle, a lightweight, general-purpose, and precision-preserving context selection technique, formulated in terms of CFL-reachability, for enabling partial context sensitivity to be adopted for the first time in object-sensitive pointer analysis algorithms. Our extensive evaluation demonstrates that with partial context sensitivity, the classic *k*-object-sensitive pointer analysis can achieve substantial performance speedups while maintaining its precision. Our Eagle-style pre-analysis is both effective and efficient, as it is formulated as a fully context-sensitive taint analysis based on CFL-reachability. It is our hope that our CFL-reachability-based approach can provide some thought-provoking insights on developing new techniques for scaling object-sensitive pointer analysis algorithms to large codebases by either preserving their precision (as demonstrated in this article) or trading precision for efficiency but with the expectation that some quantitative estimates on the precision loss can be predicted in advance.

REFERENCES

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

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 259–269. DOI: https://doi.org/10.1145/2594291.2594299

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*. ACM, New York, NY, 169–190. DOI: https://doi.org/10.1145/1167473.1167488

[4] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, New York, NY, 241–250.

[5] Cheng Cai, Qirun Zhang, Zhiqiang Zuo, Khanh Nguyen, Guoqing Xu, and Zhendong Su. 2018. Calling-to-reference context translation via constraint-guided CFL-reachability. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, 196–210.

[6] Ben-Chung Cheng and Wen-Mei W. Hwu. 2000. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*. ACM, New York, NY, 57–69. DOI: https://doi.org/10.1145/349299.349311

[7] Eagle. 2020. A Precision-Preserving Pre-Analysis Tool for Accelerating Object-Sensitive Pointer Analysis. Retrieved March 20, 2021 from http://www.cse.unsw.edu.au/~corg/eagle.

[8] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information flow analysis of Android applications in DroidSafe. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*, Vol. 15. 110.

[9] Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. 2017. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. ACM, New York, NY, 13–18.

[10] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, Los Alamitos, CA, 267–279. DOI: https://doi.org/10.1109/ASE.2019.00034

[11] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 140.

[12] Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 100.

[13] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. 2007. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems* 30, 1 (Nov. 2007), 1. DOI: https://doi.org/10.1145/1290520.1290521

[14] John Kodumal and Alex Aiken. 2004. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, NY, 207–218. DOI: https://doi.org/10.1145/996841.996867

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

[16] Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology* 18, 1 (2008), 3.

[17] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 141.

[18] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A principled approach to selective context sensitivity for pointer analysis. *ACM Transactions on Programming Languages and Systems* 42, 2 (May 2020), Article 10, 40 pages. DOI: https://doi.org/10.1145/3381915

[19] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and analyzing Java reflection. *ACM Transactions on Software Engineering and Methodology* 28, 2 (Feb. 2019), Article 7, 50 pages. DOI: https://doi.org/10.1145/3295739

[20] Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking incremental and parallel pointer analysis. *ACM Transactions on Programming Languages and Systems* 41, 1 (March 2019), Article 6, 31 pages. DOI: https://doi.org/10.1145/3293606

[21] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Communications of the ACM* 58, 2 (Jan. 2015), 44–46. DOI: https://doi.org/10.1145/2644805

[22] Jingbo Lu. 2020. *Precision-Preserving Acceleration of Object-Sensitive Pointer Analysis with CFL-Reachability*. Ph.D. Dissertation. School of Computer Science and Engineering, Faculty of Engineering, UNSW Sydney.

[23] Jingbo Lu and Jingling Xue. 2019. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), Article 148, 29 pages. DOI: https://doi.org/10.1145/3360574

[24] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An incremental points-to analysis with CFL-reachability. In *Proceedings of the 22nd International Conference on Compiler Construction (CC'13)*. 61–81. DOI: https://doi.org/10.1007/978-3-642-37051-9_4

[25] Ana Milanova. 2020. FlowCFL: A framework for type-based reachability analysis in the presence of mutable data. arXiv:2005.06496.

[26] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*. ACM, New York, NY, 1–11. DOI: https://doi.org/10.1145/566172.566174

[27] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology* 14, 1 (2005), 1–41.

[28] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. *ACM SIGPLAN Notices* 49, 6 (June 2014), 475–484. DOI: https://doi.org/10.1145/2666356.2594318

[29] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-guided program reasoning using Bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, 722–735.

[30] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *Proceedings of the Network and Distributed Security Symposium (NDSS'16)*.

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

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

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

[34] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. Carnegie Mellon University.

[35] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, New York, NY, 17–30. DOI: https://doi.org/10.1145/1926385.1926390

[36] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 485–495. DOI: https://doi.org/10.1145/2594291.2594320

[37] Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, NY, 387–400. DOI: https://doi.org/10.1145/1133981.1134027

[38] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. ACM, New York, NY, 59–76. DOI: https://doi.org/10.1145/1094811.1094817

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

[40] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: Modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, New York, NY, 278–291. DOI: https://doi.org/10.1145/3062341.3062360

[41] Rei Thiessen and OndÅej Lhoták. 2017. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, New York, NY, 263–277. DOI: https://doi.org/10.1145/3062341.3062359

[42] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective taint analysis of web applications. *ACM SIGPLAN Notices* 44, 6 (June 2009), 87–97. DOI: https://doi.org/10.1145/1543135.1542486

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

[44] Shiyi Wei and Barbara G. Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[45] Guoqing Xu, Atanas Rountev, and Manu Sridharan. 2009. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009—Object-Oriented Programming (Genoa)*. 98–122. DOI: https://doi.org/10.1007/978-3-642-03013-0_6

[46] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, New York, NY, 239–248. DOI: https://doi.org/10.1145/2594291.2594327

Received November 2020; revised January 2021; accepted February 2021