Selective Context-Sensitivity for *k-CFA* with CFL-Reachability ⋆

Jingbo Lu, Dongjie He, and Jingling Xue

Programming Languages and Compilers Group
School of Computer Science and Engineering
UNSW Sydney, Australia

Abstract. *k-CFA* provides the most well-known context abstraction for program analysis, especially pointer analysis, for a wide range of programming languages. However, its inherent context explosion problem has hindered its applicability. To mitigate this problem, selective context-sensitivity is promising as context-sensitivity is applied only selectively to some parts of the program. This paper introduces a new approach to selective context-sensitivity for supporting *k-CFA*-based pointer analysis, based on CFL-reachability. Our approach can make *k-CFA*-based pointer analysis run significantly faster while losing little precision, based on an evaluation using a set of 11 popular Java benchmarks and applications.

Keywords: Pointer Analysis · Context Sensitivity · CFL Reachability

1 Introduction

For the programs written in a wide range of programming languages, *k-CFA* [22] represents the most well-known context abstraction for representing the calling contexts of a method in program analysis, especially pointer analysis, where one remembers only the last k callsites. However, this k-callsite-based context-sensitive abstraction suffers from the combinatorial explosion of calling contexts.

For *k-CFA*-based pointer analysis, denoted kcs, different degrees of context-sensitivity at different program points in a program can have vastly different impacts on its precision and efficiency. To mitigate its context explosion problem, selective context-sensitivity is promising, since context-sensitivity is applied only selectively to some parts of the program. However, existing attempts [24,15,9,11],

⋆ Thanks to all the reviewers for their constructive comments. This work is supported by Australian Research Council Grants (DP170103956 and DP180104069).

while being applicable to other types of context-sensitivity, such as object-sensitivity [14] and type-sensitivity [23], are mostly heuristics-driven. For example, some heuristics used are related to the number of objects pointed to by some variables and certain pre-defined value-flow patterns found (according to the points-to information that is pre-computed by Andersen's (context-insensitve) analysis [3]).

In this paper, we introduce a new approach to selective context-sensitivity in order to enable kcs to run substantially faster while losing little precision. Our pre-analysis makes such selections systematically by reasoning about the effects of selective context-sensitivity on kcs, based on context-free language reachability, i.e., *CFL-reachability*. Our key insight is that the effects of analyzing a given variable/object context-sensitively on avoiding generating spurious points-to relations (elsewhere in the program) can be captured by CFL-reachability in terms of two CFLs, L_F for specifying field accesses and L_C for specifying context-sensitivity. This correlation is analytical rather than black-box-like as before. By regularizing L_F while keeping L_C unchanged, this correlation becomes efficiently verifiable (i.e., linear in terms of the number of edges in a pointer-assignment-graph representation of the program, in practice).

In summary, we make the following contributions:

- We introduce a CFL-reachability-based approach, SELECTX, for enabling selective context-sensitivity in kcs, by correlating the context-sensitivity of a variable/object selected at a program point and its effects on avoiding spurious points-to relations elsewhere (i.e., at other program points).
- We have implemented SELECTX in SOOT [32] with its source code being available at http://www.cse.unsw.edu.au/~corg/selectx.
- We have evaluated SELECTX using a set of 11 popular Java benchmarks and applications. SELECTX can boost the performance of kcs (baseline) substantially with little loss of precision. In addition, kcs also runs significantly faster for these programs overall under SELECTX than under ZIPPER [11] (an existing heuristics-based approach to selective context-sensitivity) while also being highly more precise.

The rest of this paper is organized as follows. Section 2 outlines our key insights and some challenges faced in developing SELECTX. Section 3 reviews an existing CFL-reachability formulation of kcs. Section 4 formulates SELECTX for supporting selective context-sensitivity. Section 5 evaluates SELECTX. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 Challenges

We review briefly the classic *k-CFA*-based pointer analysis (kcs) and describe some challenges faced in enabling kcs to support selective context-sensitivity.

2.1 *k-CFA*

A context-insensitive pointer analysis, such as Andersen's analysis [3], uses a single abstraction for a variable/object. In contrast, a context-sensitive pointer

analysis uses different abstractions for a variable/object for the different calling contexts of its containing method. Different flavors of context-sensitivity differ in how the calling contexts of a method (*method contexts*) and the "calling" (i.e., allocating) contexts of a heap object (*heap contexts*) are modeled. In kcs, a pointer analysis based on k-*CFA* [22], method and heap contexts are often represented by their k- and $(k-1)$-most-recent callsites, respectively [30].

Context-sensitively, if $pt(p)$ represents the points-to set of a variable/field p, the points-to relation between p and an object o that may be pointed to by p can be expressed as follows [14,23]:

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

where c is the method context (i.e., calling context) of p's containing method and c' is the heap context of o (i.e., the calling context under which o is allocated). The precision of a context-sensitive pointer analysis is measured by the context-insensitive points-to information obtained (with all the contexts dropped):

$$o \quad \in \quad \overline{pt}(p) \tag{2}$$

which can be more precise than the points-to information obtained directly by applying a context-insensitive pointer analysis (as failing to analyze each method for each context separately will conflate the effects of all calls to the method).

```
1  Object m(Object n) {
2      return n;
3  }
4
5  Object w1 = new Object(); // o1
6  Object w2 = new Object(); // o2
7  Object v1 = m(w1); // c1
8  Object v2 = m(w2); // c2
```

Fig. 1: An example for illustrating context sensitivity.

Let us consider a simple program given in Figure 1. A context-insensitive pointer analysis, such as Andersen's analysis [3], models a variable/object without distinguishing its contexts, leading to a single abstraction for variable `n`. As the analysis cannot filter out unrealizable paths, the parameters and return values of the two calls to `m()` are conflated, preventing the analysis from proving that the two calls can actually return two distinct objects. Thus, $\overline{pt}(\texttt{v1}) = \overline{pt}(\texttt{v2}) = \{\texttt{o1}, \texttt{o2}\}$, where $\texttt{o2} \in \overline{pt}(\texttt{v1})$ and $\texttt{o1} \in \overline{pt}(\texttt{v2})$ are spurious. On the other hand, $1cs$ will distinguish the two calling contexts of `m()` (by thus modeling `n` under contexts $c1$ and $c2$ with two different abstractions), yielding $pt(\texttt{v1}, [\,]) = pt(\texttt{n}, [c1]) =$

$pt(\texttt{w1}, [\,]) = \{\texttt{o1}, [\,]\}$ and $pt(\texttt{v2}, [\,]) = pt(\texttt{n}, [c2]) = pt(\texttt{w2}, [\,]) = \{\texttt{o2}, [\,]\}$, and consequently, the following more precise points-to sets, $\overline{pt}(\texttt{v1}) = \{\texttt{o1}\}$ and $\overline{pt}(\texttt{v2}) = \{\texttt{o2}\}$ (without the spurious points-to relations $\texttt{o2} \in \overline{pt}(\texttt{v1})$ and $\texttt{o1} \in \overline{pt}(\texttt{v2})$).

In this paper, we focus on context-sensitivity, as some other dimensions of precision, e.g., flow sensitivity [27,7] and path sensitivity [28,6] are orthogonal.

2.2 Selective Context-Sensitivity

Even under k-limiting, kcs must take into account all the calling contexts for a method (and consequently, all its variables/objects) at all its call sites. As a result, kcs still suffers from the context explosion problem, making it difficult to increase its precision by increasing k. However, there are situations where adding more contexts does not add more precision to kcs. To improve its efficiency and scalability, some great progress on selective context-sensitivity [24,15,9,11] has been made, with a particular focus on object-sensitivity [14] and type-sensitivity [23], two other types of context-sensitivity that are also used in analyzing object-oriented programs. However, these existing techniques (heuristics-based by nature) are not specifically tailored to kcs, often failing to deliver the full performance potentials or suffering from a great loss of precision, in practice.

Why is it hard to accelerate kcs with selective context-sensitivity efficiently without losing much precision? In Figure 1, we can see that whether `n` is context-sensitive or not ultimately affects the precision of the points-to information computed for `v1` and `v2`. However, `n` is neither `v1` nor `v2` and can be far away from both with complex field accesses on `n` via a long sequence of method calls in between. How do we know that making `n` context-sensitive can avoid some spurious points-to relations that would otherwise be generated for `v1`/`v2`?

We may attempt to resort to some heuristics-based rules regarding, e.g., the alias relations between `n` and `v1`/`v2`. However, such rules often do not admit a quantitative analysis of their sufficiency and necessity. In many cases, the effects of these rules on the efficiency and precision of kcs are unpredictable and often unsatisfactory. Can the correlation between context-sensitivity and the precision of a pointer analysis be captured to support selective context-sensitivity?

As discussed above, whether a variable/object is context-sensitive or not often does not affect its own points-to relations, but rather, the points-to relations of other variables/objects. What are then the conditions for a variable/object n to affect the points-to relation $o \in \overline{pt}(v)$, i.e., that a variable v points-to o? In general, if a points-to relation holds in a pointer analysis, then it will also hold in a less precise pointer analysis. Conversely, if a points-to relation does not hold in a pointer analysis, then it will also not hold in a more precise pointer analysis.

Let $\mathcal{A}$ be a (context-sensitive) k-*CFA*-based pointer analysis, say, kcs. Let $\mathcal{A}_{CI=\{n\}}$ be its version where only a particular variable/object n in the program is analyzed context-insensitively. A sufficient and necessary condition for requiring n to be analyzed context-sensitively (when examining n in isolation) is:

$$\exists\, o \in O, v \in V \;:\; \mathcal{A} \implies o \notin \overline{pt}(v) \quad \wedge \quad \mathcal{A}_{CI=\{n\}} \implies o \in \overline{pt}(v) \qquad (3)$$

where O is the set of objects and V is the set of variables in the program.

Consider again Figure 1. Under $\mathcal{A}$, we have $\texttt{o1} \notin \overline{pt}(\texttt{v2})$ ($\texttt{o2} \notin \overline{pt}(\texttt{v1})$), but under $\mathcal{A}_{CI=\{n\}}$, we have $\texttt{o1} \in \overline{pt}(\texttt{v2})$ and $\texttt{o2} \in \overline{pt}(\texttt{v1})$, as described in Section 2.1. According to Equation (3), there exist `o2` and `v1` in the program such that

$$\mathcal{A} \implies \texttt{o2} \notin \overline{pt}(\texttt{v1}) \quad \wedge \quad \mathcal{A}_{CI=\{\texttt{n}\}} \implies \texttt{o2} \in \overline{pt}(\texttt{v1}) \tag{4}$$

For reasons of symmetry, there also exist `o1` and `v2` in the program such that

$$\mathcal{A} \implies \texttt{o1} \notin \overline{pt}(\texttt{v2}) \quad \wedge \quad \mathcal{A}_{CI=\{\texttt{n}\}} \implies \texttt{o1} \in \overline{pt}(\texttt{v2}) \tag{5}$$

Therefore, `n` affects the points-to information computed for `v1` and `v2`. Thus, `n` should be analyzed context-sensitively in order to avoid the spurious points-to relations $\texttt{o1} \in \overline{pt}(\texttt{v2})$ and $\texttt{o2} \in \overline{pt}(\texttt{v1})$ that would otherwise be introduced. All the other variables/objects in the program can be context-insensitive as they do not affect any points-to relation computed in the entire program.

Let us modify this program by adding, `v1 = w2`, at its end (thus causing `v1` to point to not only `o1` but also `o2`). Now, if `n` is context-sensitive (under $\mathcal{A}$), we will obtain $\overline{pt}(\texttt{v1}) = \{\texttt{o1}, \texttt{o2}\}$ and $\overline{pt}(\texttt{v2}) = \{\texttt{o2}\}$. However, if `n` is context-insensitive, we will still obtain $\overline{pt}(\texttt{v1}) = \overline{pt}(\texttt{v2}) = \{\texttt{o1}, \texttt{o2}\}$ conservatively. In this case, there still exist `v2` and `o1` in the modified program such that Equation (5) holds. According to Equation (3), `n` must still be context-sensitive.

However, the condition stated in Equation 3 is impractical to validate: (1) it is computationally expensive since we need to solve $\mathcal{A}$ and $\mathcal{A}_{CI=\{n\}}$ for every variable/object n in the program, and (2) it would have rendered the whole exercise meaningless since we would have already solved $\mathcal{A}$.

In this paper, we exploit CFL-reachability to develop a necessary condition that is efficiently verifiable (i.e., linear in terms of the number of edges in the pointer assignment graph of the program) in order to approximate conservatively the set of variables/objects that require context-sensitivity. This allows us to develop a fast pre-analysis to parameterize kcs with selective context-sensitivity so that the resulting pointer analysis runs substantially more efficiently than before while suffering only a small loss of precision.

3 Preliminary

3.1 CFL-Reachability Formulation of *k-CFA*-based Pointer Analysis

CFL-Reachability [18], which is an extension of standard graph reachability, can be used to formulate a pointer analysis that operates on a graph representation of the statements in the program. In addition, a context-sensitive pointer analysis is often expressed as the intersection of two separate CFLs [18,25,31,13], with one specifying field accesses and the other specifying method calls. We leverage such a dichotomy to develop a new approach to selective context-sensitivity.

With CFL-reachability, a pointer analysis operates the *Pointer Assignment Graph (PAG)*, $G = (N, E)$, of a program, where its nodes represent the variables/objects in the program and its edges represent the value flow through

assignments. Figure 2 gives the rules for building the PAG for a Java program. For a method invocation at callsite c, $\widehat{c}$ and $\breve{c}$ represent the *entry context* and *exit context* for any callee invoked, respectively. Note that ret^m represents a unique return variable in any callee invoked. For a PAG edge, its label above indicates the kind of its associated statement and its label below indicates whether it is an *inter-context* (an edge spanning two different contexts) or *intra-context* edge (an edge spanning the same context). In particular, we shall speak of an (inter-context) *entry edge* $x \xrightarrow[\widehat{c}]{\texttt{assign}} y$, where $\widehat{c}$ is an entry context and an (inter-context) *exit edge* $x \xrightarrow[\breve{c}]{\texttt{assign}} y$, where $\breve{c}$ is an exit context. During the pointer analysis, we need to traverse the edges in G both forwards and backwards. For each edge $x \xrightarrow[c]{\ell} y$, its inverse edge is $y \xrightarrow[\overline{c}]{\overline{\ell}} x$. For a below-edge label $\widehat{c}$ or $\breve{c}$, we have $\overline{\widehat{c}} = \breve{c}$ and $\overline{\breve{c}} = \widehat{c}$, implying that the concepts of entry and exit contexts for inter-context value-flows are swapped if their associated PAG edges are traversed inversely.

A CFL-reachability-based pointer analysis makes use of two CFLs, with one being defined in terms of only above-edge labels and the other in terms of only below-edge labels [31]. Let L be a CFL over Σ formed by the above-edge (below-edge) labels in G. Each path p in G has a string $L(p)$ in Σ^* formed by concatenating in order the above-edge (below-edge) labels in p. A node v in G is said to be *L-reachable* from a node u in G if there exists a path p from u to v, known as *L-path*, such that $L(p) \in L$.

Let L_F be the CFL defined (in terms of above-edge labels) below [25,21]:

$$\begin{array}{rcl} \mathit{flowsto} & \rightarrow & \texttt{new}\ \mathit{flows}^* \\ \overline{\mathit{flowsto}} & \rightarrow & \overline{\mathit{flows}}^*\ \overline{\texttt{new}} \\ \mathit{flows} & \rightarrow & \texttt{assign} \mid \texttt{store}[f]\ \overline{\mathit{flowsto}}\ \mathit{flowsto}\ \texttt{load}[f] \\ \overline{\mathit{flows}} & \rightarrow & \overline{\texttt{assign}} \mid \overline{\texttt{load}[f]}\ \overline{\mathit{flowsto}}\ \mathit{flowsto}\ \overline{\texttt{store}[f]} \end{array} \tag{6}$$

If o *flowsto* v, then v is L_F-reachable from o. In addition, o *flowsto* v iff $v\ \overline{\mathit{flowsto}}\ o$. This means that $\overline{\mathit{flowsto}}$ actually represents the standard points-to relation. As a result, L_F allows us to perform a context-insensitive pointer analysis with CFL-reachability by solving a balanced parentheses problem for field accesses [18,26].

Let L_C be the CFL defined (in terms of below-edge labels) below [31,13]:

$$\begin{array}{rcl} \mathit{realizable} & \rightarrow & \mathit{exits}\ \mathit{entries} \\ \mathit{exits} & \rightarrow & \mathit{exits}\ \mathit{balanced} \mid \breve{c}\ \mathit{exits} \mid \epsilon \\ \mathit{entries} & \rightarrow & \mathit{balanced}\ \mathit{entries} \mid \mathit{entries}\ \widehat{c} \mid \epsilon \\ \mathit{balanced} & \rightarrow & \mathit{balanced}\ \mathit{balanced} \mid \widehat{c}\ \mathit{balanced}\ \breve{c} \mid \epsilon \end{array} \tag{7}$$

A path p in G is said to be *realizable* in the traditional sense that "returns" must be matched with their corresponding "calls" iff it is an L_C-path.

Now, kcs can be expressed by reasoning about the intersection of these two CFLs, i.e., $L_{FC} = L_F \cap L_C$. A variable v points to an object o iff there exists

Statement	**PAG Edges**
$x = \texttt{new}\ T()\ //\ o$	$o \xrightarrow{\texttt{new}} x \quad x \xrightarrow{\overline{\texttt{new}}} o$
$x = y$	$y \xrightarrow{\texttt{assign}} x \quad x \xrightarrow{\overline{\texttt{assign}}} y$
$x.f = y$	$y \xrightarrow{\texttt{store}[f]} x \quad x \xrightarrow{\overline{\texttt{store}}[f]} y$
$x = y.f$	$y \xrightarrow{\texttt{load}[f]} x \quad x \xrightarrow{\overline{\texttt{load}}[f]} y$
$x = m(..., a_i, ...)\ //\ c$	$a_i \xrightarrow[\widehat{c}]{\texttt{assign}} p_i^m \quad p_i^m \xrightarrow[\breve{c}]{\overline{\texttt{assign}}} a_i$ $ret^m \xrightarrow[\breve{c}]{\texttt{assign}} x \quad x \xrightarrow[\widehat{c}]{\overline{\texttt{assign}}} ret^m$

Fig. 2: Statements and their corresponding PAG edges.

a path p from o to v in G, such that $L_F(p) \in L_F$ (p is a *flowsto*-path) and $L_C(p) \in L_C$ (p is a realizable path). Such a path is referred to as an L_{FC}*-path*.

Note that this CFL-reachability-based formulation does not specify how the call graph is constructed. This can be either pre-computed by applying Andersen's analysis [3] or built on the fly. In Section 4, we discuss how this particular aspect of L_{FC} can cause SELECTX not to preserve the precision of kcs.

3.2 Transitivity of L_C-Path Concatenation

Given a realizable path p in G, where $L_C(p) \in L_C$, $L_C(p)$ is derived from the start symbol, *realizable*, starting with the production *realizable* $\rightarrow$ *exits entries*. Let us write $ex(p)$ for the prefix of $L_C(p)$ that is derived from *exits* and $en(p)$ for the suffix of $L_C(p)$ that is derived from *entries*. Let s be a string formed by some context labels, i.e., some below-edge labels in G. Let $can(s)$ be the canonical form of s with all balanced parentheses removed from s. For example, $can(\breve{c_1}\widehat{c_2}\breve{c_2}\widehat{c_3}) = \breve{c_1}\widehat{c_3}$. Let $str(s)$ return the same string s except that $\widehat{\ }$ or $\breve{\ }$ over each label in s has been removed. For example, $str(\breve{c_1}\widehat{c_3}) = c_1c_3$. Finally, $\overline{s}$ returns the same string s but reversed. For example, $\overline{c_1c_3} = c_3c_1$.

There exists an L_{FC}-path p from an object o to a variable v in G iff the following context-sensitive points-to relation is established:

$$(o, [str(can(ex(p)))]) \quad \in \quad pt(v, [\overline{str(can(en(p)))}]) \tag{8}$$

For brevity, we will write $scex(p)$ as a shorthand for $str(can(ex(p)))$ and $scen(p)$ as a shorthand for $\overline{str(can(en(p)))}$.

As a result, any subpath of an L_{FC}-path induces some context-sensitive points-to relations. Let $p_{x,y}$ be a path in G, starting from node x and ending at node y. Let $p_{x_1,x_2,...,x_n}$ be a path in G from node x_1 to node x_n that passes through the intermediate nodes x_2, x_3, ..., x_{n-1} in that order, which is naturally formed by $n-1$ subpaths, $p_{x_1,x_2}, p_{x_2,x_3}, ..., p_{x_{n-1},x_n}$. Consider a *flowsto*-path:

$$p_{o,n,v} = o \xrightarrow{\texttt{new}} x \xrightarrow{\texttt{store}[f]} y \xrightarrow[\widehat{c_1}]{\overline{\texttt{assign}}} z \xrightarrow{\overline{\texttt{new}}} n \xrightarrow{\texttt{new}} z \xrightarrow[\check{c_2}]{\texttt{assign}} u \xrightarrow{\texttt{load}[f]} v \tag{9}$$

Let us examine its two subpaths. For one subpath given below:

$$p_{o,n} = o \xrightarrow{\texttt{new}} x \xrightarrow{\texttt{store}[f]} y \xrightarrow[\widehat{c_1}]{\overline{\texttt{assign}}} z \xrightarrow{\overline{\texttt{new}}} n \tag{10}$$

we find that $(o, [\,]) \in pt(n.f, [c_1])$, i.e., $n.f$ under context c_1 points to o under $[\,]$, where $scex(p_{o,n}) = \epsilon$ and $scen(p_{o,n}) = c_1$. From the other subpath:

$$p_{n,v} = n \xrightarrow{\texttt{new}} z \xrightarrow[\check{c_2}]{\texttt{assign}} u \xrightarrow{\texttt{load}[f]} v \tag{11}$$

we find that $pt(n.f, [c_2]) \subseteq pt(v, [\,])$, i.e., v under $[\,]$ points to whatever $n.f$ points to under context c_2, where $scex(p_{n,v}) = c_2$ and $scen(p_{n,v}) = \epsilon$.

In general, L_C-path concatenation is not transitive. In the following theorem. we give a sufficient and necessary condition to ensure its transitivity. Given two strings s_1 and s_2, we write $s_1 \doteq s_2$ to mean that one is the prefix of the other.

Theorem 1 (Transitive L_C-Path Concatenations). *Let $p_{x,z}$ be a path in G formed by concatenating two L_C-paths, $p_{x,y}$ and $p_{y,z}$. Then $p_{x,z}$ is an L_C-path iff $scen(p_{x,y}) \doteq scex(p_{y,z})$.*

Proof. As $L_C(p_{x,y}) \in L_C$, $L_C(p_{x,y}) = ex(p_{x,y})en(p_{x,y})$ holds. Similarly, as $L_C(p_{y,z}) \in L_C$, we also have $L_C(p_{y,z}) = ex(p_{y,z})en(p_{y,z})$. As $p_{x,z}$ is formed by concatenating $p_{x,y}$ and $p_{y,z}$, we have $L_C(p_{x,z}) = ex(p_{x,y})en(p_{x,y})ex(p_{y,z})en(p_{y,z})$. To show that $p_{x,z}$ is an L_C-path, it suffices to show that $L_C(p_{x,z}) \in L_C$. According to the grammar defining L_C, $L_C(p_{x,z}) \in L_C \iff en(p_{x,y})ex(p_{y,z}) \in L_C \iff can(en(p_{x,y}))can(ex(p_{y,z})) \in L_C$, where $can(en(p_{x,y}))$ is a sequence of entry contexts $\widehat{c}$ and $can(ex(p_{y,x}))$ is a sequence of exit contexts $\check{c}$. By definition, $scen(p_{x,y}) = \overline{str(can(en(p_{x,y})))}$ and $scex(p_{y,z}) = str(can(ex(p_{y,z})))$. Thus, $can(en(p_{x,y}))can(ex(p_{y,z})) \in L_C$ iff $scen(p_{x,y}) \doteq scex(p_{y,z})$.

Let us revisit the *flowsto*-path $p_{o,n,v}$ given in Equation (9), which is a concatenation of two L_C-paths, $p_{o,n}$ and $p_{n,v}$, where $scen(p_{o,n}) = c_1$ and $scex(p_{n,v}) = c_2$. By Theorem 1, $p_{o,n,v}$ is an L_C-path iff $scen(p_{o,n}) \doteq scex(p_{n,v})$, i.e., iff $c_1 = c_2$.

4 CFL-Reachability-based Selective Context-Sensitivity

In this section, we introduce SELECTX, representing a new approach to selective context-sensitivity for accelerating kcs. Section 4.1 gives a necessary condition, which is efficiently verifiable, for making a node in G context-sensitive, based on L_{FC}. Section 4.2 describes the context-sensitivity-selection algorithm developed for SELECTX from this necessary condition for over-approximating the set of context-sensitive nodes selected. Section 4.3 explains why SELECTX may lose precision (due to the lack of provision for call graph construction in L_{FC}). Section 4.4 discusses its time and space complexities.

4.1 Necessity for Context-Sensitivity

For the *flowsto*-path $p_{o,n,v}$ given in Equation (9) discussed in Section 3.2, we can see that whether $p_{o,n,v}$ is considered to be realizable depends on whether n is modeled context-sensitively or not. We can now approximate Equation (3) in terms of CFL-reachability. Let $\mathcal{P}(G)$ be the set of paths in G. According to Theorem 1, we can conclude that a node (i.e., variable/object) n in G is context-sensitive only if the following condition holds:

$$\begin{aligned} \exists\, p_{o,n,v} \in \mathcal{P}(G)\ : & L_F(p_{o,n,v}) \in L_F \\ & \wedge L_C(p_{o,n}) \in L_C \wedge L_C(p_{n,v}) \in L_C \\ & \wedge \mathit{scen}(p_{o,n}) \neq \mathit{scex}(p_{n,v}) \end{aligned} \tag{12}$$

where o ranges over the set of objects and v over the set of variables in G.

To understand its necessity, we focus one single fixed path $p_{o,n,v} \in \mathcal{P}(G)$, where $L_F(p_{o,n,v}) \in L_F \wedge L_C(p_{o,n}) \in L_C \wedge L_C(p_{n,v}) \in L_C$ holds. By Theorem 1, if $\mathit{scen}(p_{o,n}) \neq \mathit{scex}(p_{n,v})$, we will infer $o \in \overline{pt}(v)$ spuriously when n is context-insensitive ($\mathcal{A}_{CI=\{n\}} \implies o \in \overline{pt}(v)$) but $o \notin pt(v)$ when n is context-sensitive ($\mathcal{A} \implies o \notin \overline{pt}(v)$). To understand its non-sufficiency, we note that the same object may flow to a variable along several *flowsto*-paths.

Let us consider the program in Figure 1, with its path $p_{\texttt{o1,n,v2}}$ given below:

$$\texttt{o1} \xrightarrow{\texttt{new}} \texttt{w1} \xrightarrow[\widehat{c1}]{\texttt{assign}} \texttt{n} \xrightarrow[\check{c2}]{\texttt{assign}} \texttt{v2} \tag{13}$$

where $L_F(p_{\texttt{o1,n,v2}}) \in L_F$, $L_C(p_{\texttt{o1,n}}) \in L_C$, and $L_C(p_{\texttt{n,v2}}) \in L_C$. According to Equation (12), n must be necessarily context-sensitive in order to avoid generating the spurious points-to relation $\texttt{o1} \in \overline{pt}(\texttt{v2})$ along this *flowsto*-path, since $\mathit{scen}(p_{\texttt{o1,n}}) = c1$ and $\mathit{scex}(p_{\texttt{n,v2}}) = c2$, where $c1 \neq c2$. All the other variables/objects in this program can be context-insensitive, as no such a path exists.

However, computing the CFL-reachability information according to $L_{FC} = L_F \cap L_C$ is undecidable [19]. Thus, verifying Equation (12), which is expressed in terms of L_{FC}, both efficiently and precisely for every node in G is not possible.

Below we over-approximate L_{FC} by regularizing L_F and maintaining L_C unchanged, so that the necessary condition stated in Equation (12) for a node to be context-sensitive is weakened in the new language $L_{RC} = L_R \cap L_C$.

```
1  class A {
2      Object f;
3  }
4
5  Object id(Object n) {
6      A a = new A(); // o3
7      a.f = n;
8      return a;
9  }
10
11 Object w1 = new Object(); // o1
12 Object w2 = new Object(); // o2
13 Object v1 = id(w1); // c1
14 Object v2 = id(w2); // c2
```

Fig. 3: An example for illustrating an over-approximation of L_F with L_R.

By regularizing L_F, we obtain the regular language L_R defined below:

$$\begin{array}{rcl} \mathit{flowsto} & \rightarrow & \texttt{new}\ \overline{\mathit{flows}}^{*} \\ \overline{\mathit{flowsto}} & \rightarrow & \mathit{flows}^{*}\ \overline{\texttt{new}} \\ \mathit{flows} & \rightarrow & \texttt{assign} \mid \texttt{store}\ \overline{\texttt{assign}}^{*}\ \overline{\texttt{new}}\ \texttt{new} \\ \overline{\mathit{flows}} & \rightarrow & \overline{\texttt{assign}} \mid \overline{\texttt{new}}\ \texttt{new}\ \texttt{assign}^{*}\texttt{store} \end{array} \tag{14}$$

Here, the context-insensitive pointer analysis specified by L_R is field-insensitive. In addition, loads are treated equivalently as assignments, so that `load` [_] has been replaced by `assign` and $\overline{\texttt{load}[\_]}$ by $\overline{\texttt{assign}}$. Finally, `store` [_] and $\overline{\texttt{store}[\_]}$ are no longer distinguished, so that both are now represented by `store`.

It is not difficult to see that L_R is a superset of L_F. Given $L_{RC} = L_R \cap L_C$, the necessary condition in Equation (12) can now be weakened as follows:

$$\begin{array}{ll} \exists\, p_{o,n,v} \in \mathcal{P}(G) & : L_R(p_{o,n,v}) \in L_R \\ & \quad \wedge\, L_C(p_{o,n}) \in L_C \wedge L_C(p_{n,v}) \in L_C \\ & \quad \wedge\, \mathit{scen}(p_{o,n}) \not\equiv \mathit{scex}(p_{n,v}) \end{array} \tag{15}$$

Let us illustrate this approximation by using the example given in Figure 3, which is slightly modified from the example in Figure 1. In this example, n can be context-insensitive by Equation (12) but must be context-sensitive conservatively by Equation (15). Consider the following path $p_{\texttt{o1,n,v2}}$ in G:

$$\texttt{o1} \xrightarrow{\texttt{new}} \texttt{w1} \xrightarrow[\widehat{c1}]{\texttt{assign}} \texttt{n} \xrightarrow{\texttt{store}[f]} \texttt{a} \xrightarrow{\overline{\texttt{new}}} \texttt{o3} \xrightarrow{\texttt{new}} \texttt{a} \xrightarrow[\check{c2}]{\texttt{assign}} \texttt{v2} \tag{16}$$

where $L_C(p_{\texttt{o1,n}}) \in L_C$ and $L_C(p_{\texttt{n,v2}}) \in L_C$. As $\mathit{scen}(p_{\texttt{o1,n}}) = c_1$ and $\mathit{scex}(p_{\texttt{n,v2}}) = c_2$, we have $\mathit{scen}(p_{\texttt{o1,n}}) \not\equiv \mathit{scex}(p_{\texttt{n,v2}})$. It is easy to see that $L_F(p_{\texttt{o1,n,v2}}) \notin L_F$ but

$L_R(p_{\texttt{o1,n,v2}}) \in L_R$ (with `store[f]` being treated as `store`). Thus, n should be context-sensitive according to Equation (15) but context-insensitive according to Equation (12), since $p_{\texttt{o1,n,v2}}$ is a *flowsto*-path in L_R but not in L_F.

4.2 SELECTX: a Context-Sensitivity Selection Algorithm

We describe our pre-analysis algorithm used in SELECTX for finding all context-sensitive nodes in G, by starting from the necessary condition stated in Equation (15) and then weakening it further, so that the final necessary condition becomes efficiently verifiable. We will then use this final necessary condition as if it were sufficient to find all the context-sensitive nodes in G. As a result, SELECTX may classify conservatively some context-insensitive nodes as also being context-sensitive. When performing its pre-analysis on a program, SELECTX relies on the call graph built for the program by Andersen's analysis [3]. As is standard, all the variables in a method are assumed to be in SSA form.

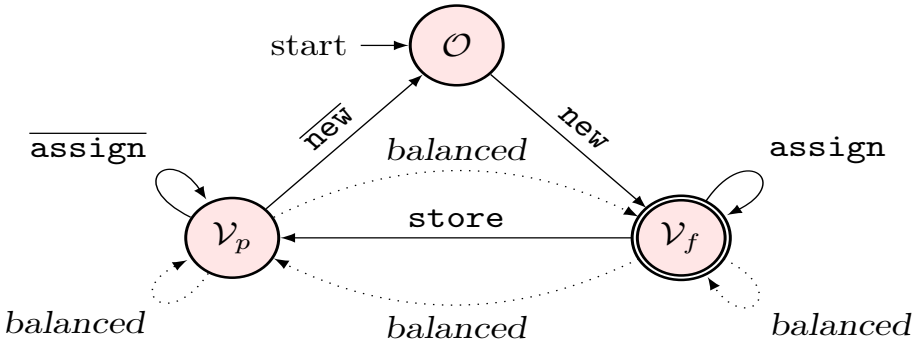

Fig. 4: A DFA for accepting L_R (with the summary edges shown in dotted lines). Note that in L_R loads are treated equivalently as assignments, so that `load` [_] is replaced by `assign` and $\overline{\texttt{load}}$[_] by $\overline{\texttt{assign}}$. In addition, `store` [_] and $\overline{\texttt{store}}$[_] are no longer distinguished, so that both are now represented by `store`.

As L_R is regular, Figure 4 gives a DFA (Deterministic Finite Automaton), A_R, for accepting this regular language over $G = (N, E)$ (the PAG of the program). There are three states: $\mathcal{S}_R = \{\mathcal{O}, \mathcal{V}_f, \mathcal{V}_p\}$, where $\mathcal{O}$ is the start state and $\mathcal{V}_f$ is the accepting state. All the objects in N must stay only in state $\mathcal{O}$. However, a variable v in N can be in state $\mathcal{V}_f$ when it participates in the computation of *flowsto* into the variable and state $\mathcal{V}_p$ when it participates in the computation of $\overline{\textit{flowsto}}$ from the variable. Therefore, $G = (N, E)$ leads naturally to a stateful version, $G_R = (N_R, E_R)$, where $N_R \subseteq N \times \{\mathcal{O}, \mathcal{V}_f, \mathcal{V}_p\}$ according to the state transitions given. For example, if $\texttt{o} \xrightarrow{\texttt{new}} \texttt{v} \in E$, then $(\texttt{o}, \mathcal{O}) \xrightarrow{\texttt{new}} (\texttt{v}, \mathcal{V}_f) \in E_R$, and if $\texttt{x} \xrightarrow{\texttt{store[\_]}} \texttt{y} \in E$, then $(\texttt{x}, \mathcal{V}_f) \xrightarrow{\texttt{store[\_]}} (\texttt{y}, \mathcal{V}_p) \in E_R$. For a pair of nodes

$n_1, n_2 \in N_R$, we write t_{n_1,n_2} if A_R can transit from n_1 to n_2 in a series of transitions and $p(t_{n_1,n_2})$ for the path traversed in G during these transitions. The four dotted summary edges (labeled *balanced*) will be discussed later.

We can now recast Equation (15) in terms of A_R. A node $n \in N_R$ (with its state label dropped) is context-sensitive only if the following condition holds:

$$\begin{aligned} \exists\, o \in N \times \{\mathcal{O}\}, v \in N \times \{\mathcal{V}_f\} \;:\; & t_{o,n} \wedge t_{n,v} \\ & \wedge L_C(p(t_{o,n})) \in L_C \wedge L_C(p(t_{n,v})) \in L_C \\ & \wedge \mathit{scen}(p(t_{o,n})) \not\equiv \mathit{scex}(p(t_{n,v})) \end{aligned} \tag{17}$$

where $t_{o,n} \wedge t_{n,v}$ holds iff the path consisting of $p(t_{o,n})$ and $p(t_{n,v})$ is a *flowsto*-path in L_R. This simplified condition can still be computationally costly (especially since SELECTX is developed as a pre-analysis), as there may be m_1 incoming paths and m_2 outgoing paths for n, resulting in $\mathit{scen}(p(t_{o,n})) \not\equiv \mathit{scex}(p(t_{n,v}))$ to be verified in $m_1 m_2$ times unnecessarily.

Instead of verifying $\mathit{scen}(p(t_{o,n})) \not\equiv \mathit{scex}(p(t_{n,v}))$ for each such possible path combination directly, we can simplify it into $\mathit{scen}(p(t_{o,n})) \neq \epsilon \wedge \mathit{scex}(p(t_{n,v})) \neq \epsilon$. As a result, we have weakened the necessary condition given in Equation (17) to:

$$\begin{aligned} \exists\, o \in N \times \{\mathcal{O}\}, v \in N \times \{\mathcal{V}_f\} \;:\; & t_{o,n} \wedge t_{n,v} \\ & \wedge L_C(p(t_{o,n})) \in L_C \wedge L_C(p(t_{n,v})) \in L_C \\ & \wedge \mathit{scen}(p(t_{o,n})) \neq \epsilon \wedge \mathit{scex}(p(t_{n,v})) \neq \epsilon \end{aligned} \tag{18}$$

This is the final necessary condition that will be used as a sufficient condition in SELECTX to determine whether n requires context-sensitivity or not. As a result, SELECTX can sometimes mis-classify a context-insensitive node as being context-sensitive. However, this conservative approach turns out to be a good design choice: SELECTX can make *kcs* run significantly faster while preserving its precision (if the precision loss caused due to the lack of provision for call graph construction in L_{FC} is ignored as discussed in Section 4.3).

SELECTX makes use of the three rules given in Figure 5 to determine that a node $n \in N_R$ (with its state label dropped) is context-sensitive if

$$n.\mathit{enflow} \neq \emptyset \quad \wedge \quad n.\mathit{exflow} \neq \emptyset \tag{19}$$

Each node $n \in N_R$ in a method m is associated with two attributes, *enflow* and *exflow*: (1) $n.\mathit{enflow}$ represents the set of sink nodes n_t of all the inter-context entry edges $n_s \underset{\widehat{c}}{\rightarrow} n_t$ of m, such that n_t can reach n, and (2) $n.\mathit{exflow}$ represents the set of source nodes n_s of all the inter-context exit edges $n_s \underset{\check{c}}{\rightarrow} n_t$ of m, such that n can reach n_s. For reasons of symmetry, both attributes are computed in exactly the same way, except that the information flows in opposite directions.

Therefore, we explain only the parts of the three rules for computing *enflow*. In [INTER-CONTEXT], we handle an inter-context edge $n \underset{\widehat{c}}{\rightarrow} n'$ by including n' in $n'.\mathit{enflow}$, i.e., initializing $n'.\mathit{enflow}$ to include also n'. In [INTRA-CONTEXT],

$$\frac{n \underset{\widehat{c}}{\rightarrow} n' \in E_R}{n' \in n'.enflow} \qquad \frac{n \underset{\widehat{c}}{\rightarrow} n' \in E_R}{n \in n.exflow} \qquad \text{[Inter-Context]}$$

$$\frac{n \rightarrow n' \in E_R}{n.enflow \subseteq n'.enflow \quad n'.exflow \subseteq n.exflow} \qquad \text{[Intra-Context]}$$

$$\frac{n \underset{\widehat{c}}{\rightarrow} n' \in E_R \quad n'' \in n.enflow \vee n \in n''.exflow \quad n''' \underset{\widehat{c}}{\rightarrow} n'' \in E_R}{n''' \xrightarrow{balanced} n' \in E_R} \qquad \text{[Callsite-Summary]}$$

Fig. 5: The rules used in Selectx for realizing selective context-sensitivity for kcs. Note that all the above-edge labels in the PAG edges are irrelevant.

we we handle an intra-context edge $n \rightarrow n'$ by simply propagating the data-flow facts from the source node n to the sink node n'. In [Callsite-Summary], we perform a standard context-sensitive summary for a callsite, as also done in the classic IFDS algorithm [20], by introducing a summary edge $n''' \xrightarrow{balanced} n'$ in $G_R = (N_R, E_R)$ to capture all possible forms of inter-procedural reachability, including, for example, a possible summary edge from an argument n''' to n' in $n' = foo(n''', ...)$, where foo is a callee method being analyzed.

To relate Equation (18) with Equation (19), we note that Selectx ensures that

(1) n always lies on a *flowsto*-paths in L_R but the path may not necessarily have to start from an object as stated by $t_{o,n} \wedge t_{n,v}$ in Equation (18) ([Inter-Context] and [Intra-Context]);
(2) $L_C(p(t_{o,n})) \in L_C \wedge L_C(p(t_{n,v})) \in L_C$ holds since Selectx performs only context-sensitive summaries ([Callsite-Summary]); and
(3) $scen(p(t_{o,n})) \neq \epsilon \wedge scex(p(t_{n,v})) \neq \epsilon$ are checked equivalently as $n.enflow \neq \emptyset \wedge n.exflow \neq \emptyset$.

Due to (1), Equation (19) is slightly weaker than Equation (18). As presented here, Selectx becomes theoretically more conservative in the sense that it may identify potentially more context-insensitive nodes as being context-sensitive. In practice, however, the differences between the two conditions are negligible, as almost all the variables in a program are expected to be well initialized (i.e., non-null) and all the objects allocated in a (factory) method are supposed to be used outside (either as a receiver object or with its fields accessed). In our implementation, however, we have followed Equation (18). Finally, global variables, which are encountered during the CFL-reachability analysis, can be handled simply by resetting its two attributes.

Let us apply these rules to the example in Figure 1, as illustrated in Figure 6. Given the two inter-context entry edges in G_R, $(\texttt{w1}, \mathcal{V}_f) \xrightarrow[\widehat{c1}]{\texttt{assign}} (\texttt{n}, \mathcal{V}_f)$

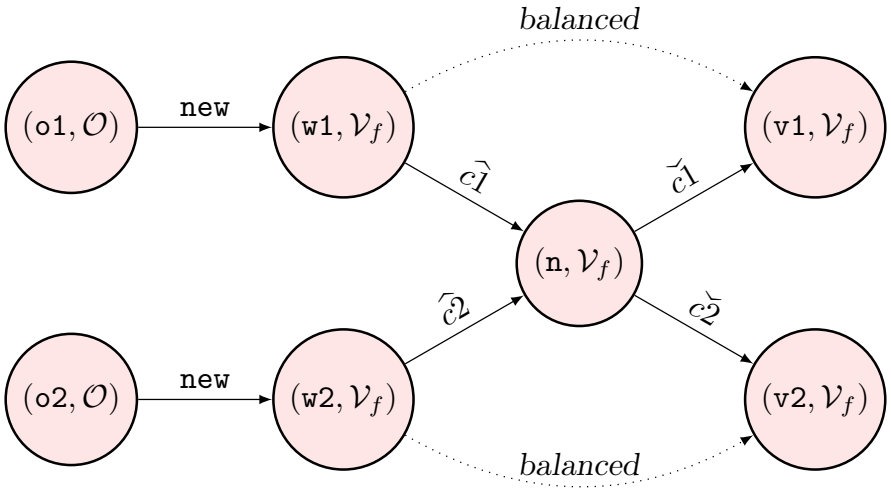

Fig. 6: Applying the rules given in Figure 5 to the example program given in Figure 1 (represented by its stateful PAG).

and $(\texttt{w2}, \mathcal{V}_f) \xrightarrow[\widehat{c2}]{\texttt{assign}} (\texttt{n}, \mathcal{V}_f)$, we have $(n, \mathcal{V}_f).\mathit{enflow} = \{(n, \mathcal{V}_f)\}$ by [InterContext]. Due to the two inter-context exit edges, $(\texttt{n}, \mathcal{V}_f) \xrightarrow[\check{c1}]{\texttt{assign}} (\texttt{v1}, \mathcal{V}_f)$ and $(\texttt{n}, \mathcal{V}_f) \xrightarrow[\check{c2}]{\texttt{assign}} (\texttt{v2}, \mathcal{V}_f)$, we have $(\texttt{w1}, \mathcal{V}_f) \xrightarrow{balanced} (\texttt{v1}, \mathcal{V}_f)$ and $(\texttt{w2}, \mathcal{V}_f) \xrightarrow{balanced} (\texttt{v2}, \mathcal{V}_f)$ by [Callsite-Summary]. Similarly, we obtain $(\texttt{n}, \mathcal{V}_f).\mathit{exflow} = \{(\texttt{n}, \mathcal{V}_f)\}$. Thus, n is context-sensitive by Equation (19). For this example program, all the other nodes can be deduced to be context-insensitive.

4.3 Precision Loss

As Selectx selects a node in G to be context-sensitive by using the necessary condition given in Equation (18) as a sufficient condition, it is then expected to always over-approximate the set of context-sensitive nodes in G, and consequently, to always preserve the precision of any *k-CFA*-based pointer analysis algorithm like *kcs* being accelerated. However, this is not the case, as Selectx may cause the pointer analysis to sometimes suffer from a small loss of precision due to the lack of provision in L_{FC} on how the call graph for the program being analyzed should be constructed *during the CFL-reachability-based pointer analysis*.

To illustrate this situation, let us consider an example given in Figure 7. Class `A` defines two methods, `foo()` and `bar()`. Its subclass `B` inherits `foo()` from class `A` but overrides `bar()`. Note that `foo()` can be regarded as a wrapper method for `bar()`, as `foo()` simply invokes `bar()` on its "`this`" variable at line 4. If "`this`" is analyzed context-insensitively by *kcs*, then $pt(\texttt{this}, [\,]) = \{(\texttt{o1}, [\,]), (\texttt{o3}, [\,])\}$.

```
1  class A {
2      void bar(Object p) {}
3      void foo(Object s) {
4          this.bar(s); // c3
5      }
6  }
7
8  class B extends A {
9      void bar(Object q) {}
10 }
11
12 A a = new A(); // o1
13 Object v1 = new Object(); // o2
14 a.foo(v1); // c1
15 A b = new B(); // o3
16 Object v2 = new Object(); // o4
17 b.foo(v2); // c2
```

Fig. 7: Illustrating precision loss in kcs with selective context-sensitivity by SELECTX due to a missing call graph construction mechanism built-into L_{FC}.

This will cause the two `bar()` methods to be analyzed at line 4 whenever `foo()` is analyzed from each of its two callsites at lines 14 and 17, resulting in the conflation of `o2` and `o4` passed into the two `bar()` methods via their parameters at the two callsites. As a result, the two spurious points-to relations $\texttt{o2} \in \overline{pt}(\texttt{q})$ and $\texttt{o4} \in \overline{pt}(\texttt{p})$ will be generated. However, if the "`this`" variable is analyzed context-sensitively, then the two spurious points-to relations will be avoided.

If we apply Equation (12) to "`this`" at line 4, we will conclude that "`this`" is context-insensitive due to the existence of only two paths passing through "`this`" in G: $\texttt{o1} \xrightarrow{\texttt{new}} \texttt{a} \xrightarrow[\widehat{c1}]{\texttt{assign}} \texttt{this}$ and $\texttt{o3} \xrightarrow{\texttt{new}} \texttt{a} \xrightarrow[\widehat{c2}]{\texttt{assign}} \texttt{this}$. Despite the value-flows that can continue from "`this`" along the two paths in the program, L_{FC} itself is "not aware" of such value-flows, as it does not have a call-graph-construction mechanism built into its underlying CFL reachability formulation. How to fill this gap in L_{FC} will be an interesting research topic. Currently, SELECTX may select some variables representing receivers (like "`this`" at line 4) to be analyzed context-insensitively, resulting in a small loss of precision (Section 5). Note that this problem does not exist in L_{FC}-based demand-driven pointer analysis algorithms [18,25], as they combine L_{FC} and a separate call-graph construction mechanism to compute the points-to information.

4.4 Time and Space Complexities

Time Complexity. Consider a program $\mathcal{P}$ with its PAG being $G = (N, E)$ and its corresponding stateful version being $G_R = (N_R, E_R)$, where $|N_R| < 2|N|$ and $|E_R| = |E|$. Let P be the maximum number of parameters (including "this" and the return variable) in a method in $\mathcal{P}$. For each node n, we have $|n.enflow| \leqslant P$ and $|n.exflow| \leqslant P$. Thus, the worst-case time complexity for handling all the intra-context edges in $\mathcal{P}$ is $O(P \times |E|)$. As for the inter-context edges, let M be the number of methods in $\mathcal{P}$ and I be the maximum number of inter-context entry edges per method in G (i.e., G_R). In [CALLSITE-SUMMARY]), $n'' \in n.enflow$ and $n \in n''.exflow$ can each be checked in $O(P)$, since $|n.enflow| = |n.exflow| = O(P)$. Thus, the worst-case time complexity for producing all the summary edges for $\mathcal{P}$ is $O(P \times M \times I)$. Finally, the worst-case time complexity for SELECTX is $O(P \times |E| + P \times M \times I)$, which should simplify to $O(|E|)$, since (1) $P \ll |E|$, and (2) $M \times I$ represents the number of inter-context entry edges in E.

For real-world programs, their PAGs are sparse. $|E|$ is usually just several times larger than $|N|$ rather than $O(|N|^2)$, making SELECTX rather lightweight.

Space Complexity. The space needed for representing G or G_R is $O(|N|+|E|)$. The space needed for storing the two attributes, *enflow* and *exflow*, at all the nodes in G_R is $O(|N| \times P)$. Thus, the worst-case space complexity for SELECTX is $O(|N| + |E| + |N| \times P)$, which simplies to $O(|N| + |E|)$.

5 Evaluation

Our objective is to demonstrate that SELECTX can boost the performance of kcs (i.e., *k-CFA*-based pointer analysis) with only little loss of precision. To place this work in the context of existing research efforts to selective context-sensitivity [24,15,9,11], we also compare SELECTX with ZIPPER [11], a state-of-the-art pre-analysis that can also support selective context-sensitivity in kcs.

We conduct our evaluation using Java programs in SOOT [32], a program analysis and optimization framework for Java programs, as we expect our findings to carry over to the programs written in other languages. We have implemented kcs (the standard *k-CFA*-based pointer analysis) based on SPARK [10] (a context-insensitive Andersen's analysis [3]) provided in SOOT. We have also implemented SELECTX in SOOT, which performs its pre-analysis for a program based on the call graph pre-computed for the program by SPARK. For ZIPPER, we use its open-source implementation [11], which performs its pre-analysis for a program based on the points-to information pre-computed by SPARK.

We have used a set of 11 popular Java programs, including eight benchmarks from the DaCapo Benchmark suite (v.2006-10-MR2) [4] and three Java applications, `checkstyle`, `findbugs` and `JPC`, which are commonly used in evaluating pointer analysis algorithms for Java [30,29,24,9,8,13]. For DaCapo, `lusearch` is excluded as it is similar to `luindex`. We have also excluded `bloat` and `jython`, as $3cs$ fails to scale either by itself or when accelerated by any pre-analysis.

We follow a few common practices adopted in the recent pointer analysis literature [1,16,17,2,30,11,13]. We resolve Java reflection by using a dynamic reflection analysis tool, TamiFlex [5]. For native code, we make use of the method summaries provided in Soot. String- and Exception-like objects are distinguished per dynamic type and analyzed context-insensitively. For the Java Standard Library, we have used `JRE1.6.0_45` to analyze all the 11 programs.

We have carried out all our experiments on a Xeon E5-1660 3.2GHz machine with 256GB of RAM. The analysis time of a program is the average of 3 runs.

Table 1 presents our results, which will be analyzed in Sections 5.1 and 5.2. For each $k \in \{2, 3\}$ considered, kcs is the baseline, z-kcs is the version of kcs accelerated by Zipper, and s-kcs is the version of kcs accelerated by Selectx. The results for Spark (denoted ci) are also given for comparison purposes.

As is standard nowadays, kcs computes the points-to information for a program by constructing its call graph on the fly. As for exception analysis, we use Spark's its built-in mechanism for handling Java exceptions during the pointer analysis. We wish to point out that how to handle exceptions does not affect the findings reported in our evaluation. We have also compared s-kcs with ci, kcs and z-kcs for all the 11 programs by ignoring their exception-handling statements (i.e., throw and catch statements). For each program, the precision and efficiency ratios obtained by all the four analyses are nearly the same

5.1 *kcs* vs. s-*kcs*

In this section, we discuss how Selectx (developed based on CFL-reachability) can improve the efficiency of kcs with little loss of precision.

Efficiency. We measure the efficiency of a pointer analysis in terms of the analysis time elapsed in analyzing a program to completion. The time budget set for analyzing a program is 24 hours. For all the metrics, smaller is better.

For k = 2, s-2cs outperforms 2cs by 11.2x, on average. The highest speedup achieved is 23.5x for `hsqldb`, for which 2cs spends 244.2 seconds while s-2cs has cut this down to only 10.4 seconds. The lowest speedup is 3.2x for `findbugs`, which is the second most time-consuming program to analyze by 2cs. For this program, Selectx has managed to reduce 2cs's analysis time from 1007.1 seconds to 286.2 seconds, representing an absolute reduction of 720.9 seconds.

For $k = 3$, 3cs is unscalable for all the 11 programs, but Selectx has succeeded in enabling 3cs to analyze them scalably, in under 31 minutes each.

Precision. We measure the precision of a context-sensitive pointer analysis as in [25,30,24], in terms of three metrics, which are computed in terms of the final context-insensitive points-to information obtained ($\overline{pt}$). These are "#Call Edges" (number of call graph edges discovered), "#Poly Calls" (number of polymorphic calls discovered), and "#May-Fail Casts" (number of type casts that may fail).

For each precision metric m, let ci_m, $Baseline_m$ and P_m be the results obtained by Spark, $Baseline$ and P, respectively, where $Baseline$ and P are two

Table 1: Performance and precision of kcs, z-kcs (kcs accelerated by Zipper) and s-kcs (kcs accelerated by Selectx). The results for ci (i.e., Spark) are also included for comparison purposes. For all the metrics, smaller is better.

Program	Metrics	ci	2cs	z-2cs	s-2cs	3cs	z-3cs	s-3cs
antlr	Time (secs)	7.1	277.4	11.8	18.1	OoT	51.0	57.3
	#Call Edges	56595	54212	54366	54212	-	53768	53233
	#Poly Calls	1974	1861	1879	1861	-	1823	1792
	#May-Fail Casts	1140	841	862	847	-	794	665
chart	Time (secs)	10.7	712.4	94.4	113.7	OoM	2903.6	1832.9
	#Call Edges	75009	71080	71444	71080	-	70718	69579
	#Poly Calls	2462	2290	2314	2290	-	2213	2159
	#May-Fail Casts	2252	1819	1869	1825	-	1813	1693
eclipse	Time (secs)	7.0	442.5	45.1	51.1	OoM	748.2	621.8
	#Call Edges	55677	52069	52142	52071	-	51711	50878
	#Poly Calls	1723	1551	1563	1552	-	1507	1473
	#May-Fail Casts	1582	1237	1260	1250	-	1231	1135
fop	Time (secs)	5.4	321.1	16.2	20.1	OoM	84.7	96.8
	#Call Edges	39653	37264	37457	37264	-	36809	36416
	#Poly Calls	1206	1082	1106	1082	-	1038	1005
	#May-Fail Casts	919	637	658	643	-	620	573
hsqldb	Time (secs)	5.7	244.2	7.3	10.4	OoM	29.2	42.4
	#Call Edges	40714	37565	37751	37565	-	37087	36637
	#Poly Calls	1188	1065	1091	1065	-	1032	996
	#May-Fail Casts	902	635	657	641	-	616	566
luindex	Time (secs)	6.0	238.0	7.4	10.3	OoM	29.6	46.4
	#Call Edges	38832	36462	36649	36462	-	36051	35522
	#Poly Calls	1269	1144	1168	1144	-	1112	1075
	#May-Fail Casts	921	648	671	654	-	631	574
pmd	Time (secs)	9.2	980.3	266.6	286.8	OoM	3693.6	1374.3
	#Call Edges	69148	65877	66053	65877	-	65519	64402
	#Poly Calls	2965	2782	2798	2782	-	2729	2608
	#May-Fail Casts	2293	1941	1962	1948	-	1901	1778
xalan	Time (secs)	5.6	269.3	14.0	18.4	OoT	71.8	76.0
	#Call Edges	44061	40645	40800	40645	-	40189	39756
	#Poly Calls	1394	1260	1279	1260	-	1223	1192
	#May-Fail Casts	1063	742	763	748	-	724	664
checkstyle	Time (secs)	9.7	1103.0	354.7	317.2	OoT	6851.2	852.7
	#Call Edges	79808	74792	74962	74792	-	74367	73236
	#Poly Calls	2754	2564	2583	2564	-	2519	2492
	#May-Fail Casts	1943	1549	1573	1555	-	1519	1393
findbugs	Time (secs)	10.7	1007.1	286.2	317.0	OoT	5292.9	1806.7
	#Call Edges	82958	77133	77159	77133	-	76383	75421
	#Poly Calls	3378	3043	3047	3043	-	2952	2943
	#May-Fail Casts	2431	1972	2021	1988	-	1955	1821
JPC	Time (secs)	10.4	484.2	70.7	87.6	OoM	1085.5	677.9
	#Call Edges	71657	67989	68136	67999	-	67359	66060
	#Poly Calls	2889	2667	2693	2669	-	2617	2539
	#May-Fail Casts	2114	1658	1696	1666	-	1637	1543

Table 2: The analysis times of SELECTX (secs).

Program	antlr	chart	eclipse	fop	hsqldb	luindex	pmd	xalan	checkstyle	findbugs	JPC
SELECTX	10.6	33.1	12.7	9.1	7.5	8.4	28.7	10.2	31.0	34.7	12.0

context-sensitive pointer analyses such that P is less precise than *Baseline*. We measure the precision loss of P with respect to *Baseline* for this metric by

$$\frac{\big((ci_m - Baseline_m) - (ci_m - P_m)\big)}{(ci_m - Baseline_m)}.$$

Here, the precision improvement going from *ci* to *Baseline* is regarded as 100%. If $P_m = Baseline_m$, then P loses no precision at all. On the other hand, if $P_m = ci_m$, then P loses all the precision gained by *Baseline*.

Given the significant speedups obtained, SELECTX suffers from only small increases in all the three metrics measured across the 11 programs due to its exploitation of CFL-reachability for making its context-sensitivity selections. On average, the precision loss percentages for "#Call Edges", "#Poly Calls", and "#May-Fail Casts" are only 0.03%, 0.13%, and 2.21%, respectively.

Overhead. As a pre-analysis, SELECTX relies on SPARK (i.e., Andersen's pointer analysis [3]) to build the call graph for a program to facilitate its context-sensitivity selections. As shown in Tables 1 and 2, both SPARK (i.e., a context-insensitive pointer analysis (*ci*)) and SELECTX are fast. The pre-analysis times spent by SELECTX are roughly proportional to the pointer analysis times spent by SPARK across the 11 programs. SELECTX is slightly slower than SPARK but can analyze all the 11 programs in under 40 seconds each.

The overall overheads from both SPARK and SELECTX for all the 11 programs are negligible relative to the analysis times of kcs. In addition, these overheads can be amortized. The points-to information produced by SPARK is often reused for some other purposes (e.g., used by both ZIPPER and SELECTX here), and the same pre-analysis performed by SELECTX can often be used to accelerate a range of *k-CFA*-based pointer analysis algorithms (e.g., $k \in \{2, 3\}$ here).

5.2 z-kcs vs. s-kcs

We compare SELECTX with ZIPPER [11] in terms of the efficiency and precision achieved for supporting selective context-sensitivity in kcs. ZIPPER selects the set of methods in a program that should be analyzed context-sensitively and can thus be applied to not only kcs but also other types of context-sensitivity, such as object-sensitivity [14] and type-sensitivity [23]. SELECTX is, however, specifically designed for supporting kcs i.e., callsite-sensitivity.

Table 1 contains already the results obtained for z-kcs and s-kcs. Figure 8 compares and contrasts all the 5 analyses, 2cs, z-2cs, s-2cs, z-3cs, and s-3cs, in terms of their efficiency and precision across all the 11 programs (normalized

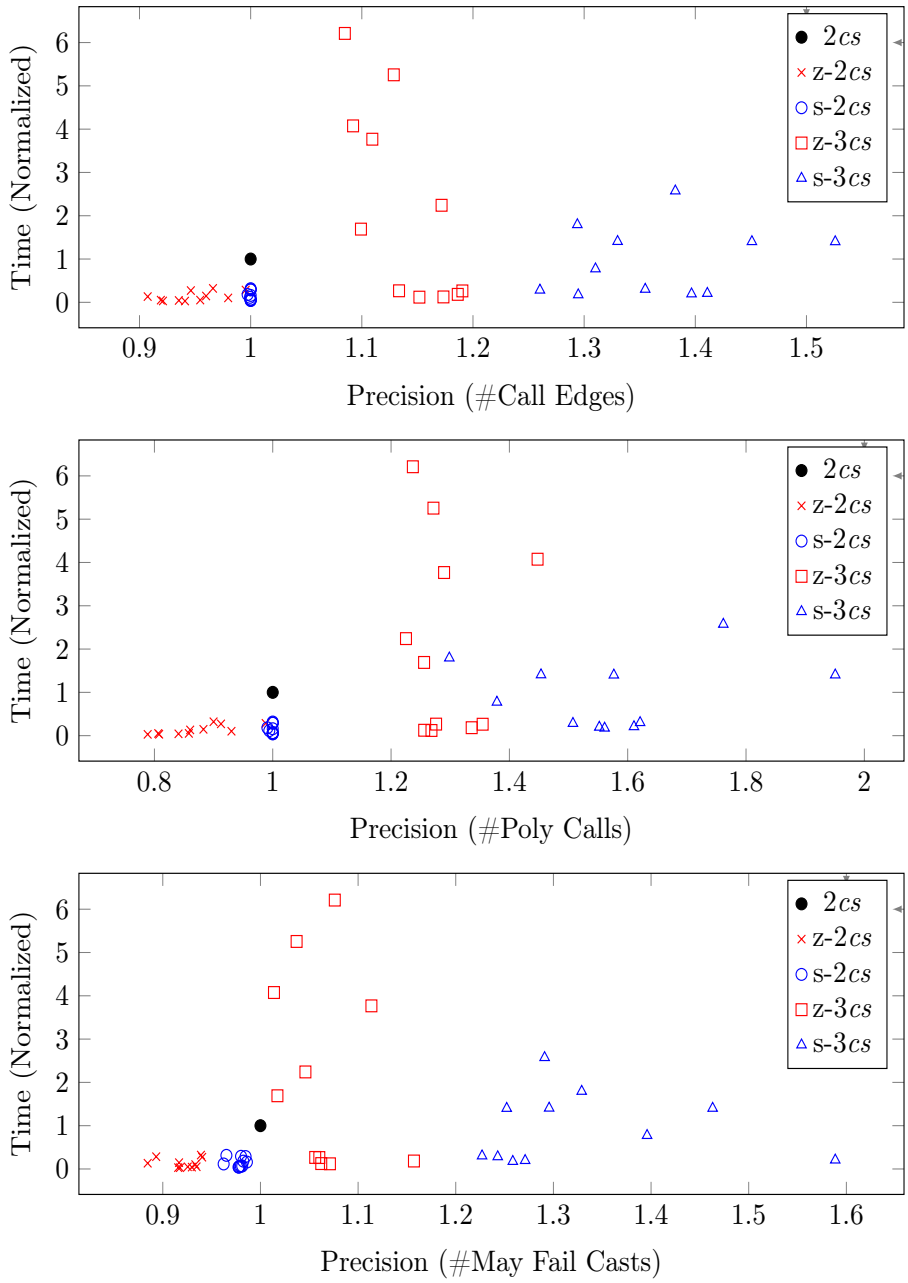

Fig. 8: Understanding the efficiency and precision of the five analyses by using the results obtained from Table 1 for all the 11 programs (normalized to $2cs$). How these scatter graphs are plotted is explained precisely in Section 5.2. For each data point plotted, the lower and further to the right, the better.

to $2cs$), in three scatter graphs, one per precision metric used. Note that $3cs$ is unscalable for all the 11 programs. For each metric m, let ci_m and A_m, where $A \in \{2cs, \text{z-}2cs, \text{s-}2cs, \text{z-}3cs, \text{s-}3cs\}$ be the results obtained by SPARK and A, respectively. If m is "analysis time", then A is plotted at $A_m/2cs_m$ along the y-axis. If $m \in$ {"#Call Edges","#Poly Calls","#May-Fail Casts"}, then A is plotted at $(ci_m - A_m)/(ci_m - 2cs_m)$ along the x-axis. Hence, $2cs$ appears at (1,1) (highlighted with a fat dot •) in all the three graphs. Here, $ci_m - A_m$ represents the absolute number of spurious call edges/poly calls/may-fail casts removed by A relative to ci. Therefore, when comparing z-kcs and s-kcs, for each $k \in \{2, 3\}$, the one that is lower is better (in terms of efficiency) and the one that is further to the right is better (in terms of a precision metric).

As discussed in Section 5.1, s-$2cs$ enables $2cs$ to achieve remarkable speedups (with ○'s appearing below •) at only small decreases in precision (with ○'s appearing very close to the left of •) with the percentage reductions being 0.03%, 0.13%, and 2.21% for "#Call Edges", "#Poly Calls", and "#May-Fail Casts", respectively, on average. On the other hand, z-$2cs$ is slightly faster than s-$2cs$ (by 1.2x, on average), but its percentage precision reductions over $2cs$ are much higher, reaching 5.22%, 12.94%, and 7.91% on average (with ×'s being further away from • to the left).

For $k = 3$, $3cs$ is not scalable for all the 11 programs. Both ZIPPER and SELECTX have succeeded in making it scalable with selective context-sensitivity for all the 11 programs. However, s-$3cs$ is not only faster than z-$3cs$ (by 2.0x on average) but also more precise (with z-$3cs$ exhibiting the percentage precision reductions of 16.38%, 16.96%, and 19.57% for "#Call Edges", "#Poly Calls", and "#May-Fail Casts", respectively, on average, relative to s-$3cs$). For five out of the 11 programs, z-$3cs$ is faster than s-$3cs$, but each of these five programs can be analyzed by each analysis in less than 2 minutes. However, s-$3cs$ is faster than z-$3cs$ for the remaining six programs, which take significantly longer to analyze each. In this case, s-$3cs$ outperforms z-$3cs$ by 3.0x, on average. A program worth mentioning is `checkstyle`, where s-$2cs$ spends 12% more analysis time than z-$2cs$, but s-$3cs$ is 8.0x faster than z-$3cs$ due to its better precision achieved.

6 Related Work

This paper is the first to leverage CFL reachability to accelerate kcs with selective context-sensitivity. There are some earlier general-purpose attempts [24,11,15,9] that can also be used for accelerating kcs with selective context-sensitivity.

Like SELECTX, "Introspective Analysis" [24] and ZIPPER [11] rely on performing a context-insensitive pointer analysis to guide their context-sensitivity selections. "Introspective Analysis" prevents some "bad" parts of a program from being analyzed context-sensitively based on some empirical threshold-based indicators. ZIPPER identifies so-called "precision-critical" methods by using classes as basic units to probe the flow of objects. In this paper, we have compared SELECTX with ZIPPER in terms of their effectiveness for improving kcs.

"Data-Driven Analysis" [9] is developed based on similar observations as in this paper: the effects of a node's context-sensitivity on the precision of a context-sensitive pointer analysis can be observed from its impact on the precision of a client analysis such as may-fail-casting. It applies machine learning to learn to make context-sensitivity selections. In contrast, SELECTX relies CFL-reachability instead without having to resort to an expensive machine learning process.

In [15], the authors also leverage a pre-analysis to decide whether certain call-sites require context-sensitivity. However, unlike the three techniques [24,11,9] discussed above and SELECTX, which rely on a context-insensitive pointer analysis to make their context-sensitivity selections, their paper achieves this by using a program analysis that is fully context-sensitive yet greatly simplified.

There are also other research efforts on accelerating kcs. MAHJONG [30] improves the efficiency of kcs by merging type-consistent allocation sites, targeting type-dependent clients, such as call graph construction, devirtualization and may-fail casting, but at the expense of alias relations. In [25], the authors accelerate a demand-driven k-*CFA*-based pointer analysis (formulated in terms of CFL-reachability) by adapting the precision of field aliases with a client's need.

EAGLE [13,12] represents a CFL-reachability-based pre-analysis specifically designed for supporting selective object-sensitivity [14] with no loss of precision. The pre-analyses discussed earlier [24,11,9] for parameterizing context-sensitive pointer analysis are also applicable to object-sensitivity.

7 Conclusion

We have introduced SELECTX, a new CFL-reachability-based approach that is specifically designed for supporting selective context-sensitivity in k-*CFA*-based pointer analysis (kcs). Our evaluation demonstrates that SELECTX can enable kcs to achieve substantial speedups while losing little precision. In addition, SELECTX also compares favorably with a state-of-the-art approach that also supports selective context-sensitivity. We hope that our investigation on CFL-reachability can provide some insights on developing new techniques for scaling kcs further to large codebases or pursuing some related interesting directions.

References

1. Ali, K., Lhoták, O.: Application-only call graph construction. In: Noble, J. (ed.) ECOOP 2012 – Object-Oriented Programming. pp. 688–712. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
2. Ali, K., Lhoták, O.: Averroes: Whole-program analysis without the whole program. In: Castagna, G. (ed.) ECOOP 2013 – Object-Oriented Programming. pp. 378–400. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
3. Andersen, L.O.: Program analysis and specialization for the C programming language. Ph.D. thesis (1994)
4. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.,

Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications. pp. 169–190. ACM Press, New York, NY, USA (Oct 2006). https://doi.org/http://doi.acm.org/10.1145/1167473.1167488
5. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 241–250. ACM (2011)
6. Bodík, R., Anik, S.: Path-sensitive value-flow analysis. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 237–251. POPL '98, Association for Computing Machinery, New York, NY, USA (1998). https://doi.org/10.1145/268946.268966
7. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. p. 289–298. CGO '11, IEEE Computer Society, USA (2011)
8. Jeon, M., Jeong, S., Oh, H.: Precise and scalable points-to analysis via data-driven context tunneling. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 140 (2018)
9. Jeong, S., Jeon, M., Cha, S., Oh, H.: Data-driven context-sensitivity for points-to analysis. Proceedings of the ACM on Programming Languages **1**(OOPSLA), 100 (2017)
10. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using spark. In: International Conference on Compiler Construction. pp. 153–169. Springer (2003)
11. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: A principled approach to selective context sensitivity for pointer analysis. ACM Trans. Program. Lang. Syst. **42**(2) (May 2020). https://doi.org/10.1145/3381915
12. Lu, J., He, D., Xue, J.: Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. ACM Transactions on Software Engineering and Methodology **30**(4) (2021), to appear.
13. Lu, J., Xue, J.: Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. Proc. ACM Program. Lang. **3** (Oct 2019). https://doi.org/10.1145/3360574
14. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Transactions on Software Engineering and Methodology (TOSEM) **14**(1), 1–41 (2005)
15. Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. SIGPLAN Not. **49**(6), 475–484 (Jun 2014). https://doi.org/10.1145/2666356.2594318
16. Raghothaman, M., Kulkarni, S., Heo, K., Naik, M.: User-guided program reasoning using bayesian inference. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 722–735. ACM (2018)
17. Rasthofer, S., Arzt, S., Miltenberger, M., Bodden, E.: Harvesting runtime values in Android applications that feature anti-analysis techniques. In: NDSS (2016)
18. Reps, T.: Program analysis via graph reachability. Information and software technology **40**(11-12), 701–726 (1998)
19. Reps, T.: Undecidability of context-sensitive data-dependence analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(1), 162–186 (2000)

20. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 49–61. POPL '95, Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/199448.199462
21. Shang, L., Xie, X., Xue, J.: On-demand dynamic summary-based points-to analysis. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization. pp. 264–274. ACM (2012)
22. Shivers, O.: Control-flow analysis of higher-order languages. Ph.D. thesis, Citeseer (1991)
23. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding object-sensitivity. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 17–30. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926390
24. Smaragdakis, Y., Kastrinis, G., Balatsouras, G.: Introspective analysis: Context-sensitivity, across the board. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 485–495. PLDI '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594320
25. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 387–400. PLDI '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1133981.1134027
26. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-driven points-to analysis for Java. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 59–76. OOPSLA '05, ACM, New York, NY, USA (2005). https://doi.org/10.1145/1094811.1094817
27. Sui, Y., Di, P., Xue, J.: Sparse flow-sensitive pointer analysis for multithreaded programs. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. p. 160–170. CGO '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2854038.2854043
28. Sui, Y., Ye, S., Xue, J., Yew, P.: SPAS: scalable path-sensitive pointer analysis on full-sparse SSA. In: Yang, H. (ed.) Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings. Lecture Notes in Computer Science, vol. 7078, pp. 155–171. Springer (2011). https://doi.org/10.1007/978-3-642-25318-8_14
29. Tan, T., Li, Y., Xue, J.: Making k-object-sensitive pointer analysis more precise with still k-limiting. In: International Static Analysis Symposium. pp. 489–510. Springer (2016)
30. Tan, T., Li, Y., Xue, J.: 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. p. 278–291. PLDI 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062360
31. Thiessen, R., Lhoták, O.: Context transformations for pointer analysis. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 263–277. PLDI 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062359

32. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: A Java bytecode optimization framework. In: CASCON First Decade High Impact Papers. pp. 214–224. IBM Corp. (2010)