Selective X-Sensitive Analysis Guided by Impact Pre-Analysis

Hakjoo Oh, Korea University
Wonchan Lee, Stanford University
Kihong Heo, Seoul National University
Hongseok Yang, University of Oxford
Kwangkeun Yi, Seoul National University

We present a method for selectively applying context-sensitivity during interprocedural program analysis. Our method applies context-sensitivity only when and where doing so is likely to improve the precision that matters for resolving given queries. The idea is to use a pre-analysis to estimate the impact of context-sensitivity on the main analysis's precision, and to use this information to find out when and where the main analysis should turn on or off its context-sensitivity. We formalize this approach and prove that the analysis always benefits from the pre-analysis-guided context-sensitivity. We implemented this selective method for an existing industrial-strength interval analyzer for full C. The method reduced the number of (false) alarms by 24.4%, while increasing the analysis cost by 27.8% on average.

The use of the selective method is not limited to context-sensitivity. We demonstrate this generality by following the same principle and developing a selective relational analysis and a selective flow-sensitive analysis. Our experiments show that the method cost-effectively improves the precision in the these analyses as well.

Categories and Subject Descriptors: F.3.2 [**Semantics of Programming Languages**]: Program Analysis

General Terms: Programming Languages, Program Analysis

Additional Key Words and Phrases: Static analysis, context-sensitive analysis, abstract interpretation

ACM Reference Format:
ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 44 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Handling procedure calls in static analysis with a right balance between precision and cost is challenging. To precisely analyze procedure calls and returns, the analysis has to distinguish calls to the same procedure by their different calling contexts. However, a simple-minded, uniform context-sensitivity at all call sites easily makes the resulting analysis non cost-effective. For example, imagine a program analysis for proving the safety of array accesses that uses the k-callstring approach [Sharir and Pnueli 1981; Shivers 1991] for abstracting calling contexts. The k-callstring approach distinguishes two calls to the same procedure whenever their k-most recent call sites are different. To make this context-sensitive analysis cost-effective, we need to tune the k values at the call sites in a way that we should increase the k value only where the increased precision contributes to the proof of array-access safety. If we simply use the same fixed k for all the call sites, the analysis would end up becoming either unnecessarily precise and costly, or not precise enough to prove the safety of many array accesses.

In this paper, we present a method for performing *selective context-sensitive* analysis which applies the context-sensitivity only when and where doing so is likely to improve the precision that matters for the analysis's ultimate goal. Our method works for callstrings-based context-sensitivity[1], and it consists of two steps. The first step is a pre-analysis that estimates the behavior of the main analysis under the full context-sensitivity (i.e. using ∞-callstrings). The pre-analysis focuses only on estimating the impact of context-sensitivity on the main analysis. Hence, it aggressively abstracts the other semantic aspects of the main analysis. The second step is the main analysis with selective context-sensitivity. This analysis uses the results of the pre-analysis, selects influential call sites for precision, and selectively applies context-sensitivity only to these call sites.

One important feature of our method is that the pre-analysis-guided context-sensitivity pays off at the subsequent selective context-sensitive analysis. One way to see the subtlety of this impact realization is to note that the pre-analysis and the selective main analysis are incomparable in precision: the pre-analysis is more precise than the main analysis in terms of context-sensitivity, but it is worse than the main analysis in tracking individual program statements. Despite this mismatch, our guidelines for designing an impact pre-analysis and the resulting selective context-sensitivity ensure that the selective context-sensitive main analysis is at least as precise as the fully context-sensitive pre-analysis, as far as given queries are concerned.

We have implemented our method on an existing industrial-strength interval analyzer for full C. The method led to the reduction of alarms between 6.6 and 48.3%, with average 24.4%, compared with the baseline context-insensitive analysis, while increasing the analysis cost between 9.4 and 50.5%, with average 27.8%.

The general principle behind the design and the use of impact pre-analysis is not limited to context-sensitivity. Instead of designing a particular selective analysis for context-sensitivity, we provide a guideline for designing impact pre-analyses for a range of static analyses. Also, we follow the same guideline to develop two other selective program analyses in order to demonstrate the applicability of the general principle. The first one is a selective relational analysis that keeps track of relationships between variables, only when tracking them are likely to help the main analysis answer given queries. In this case, the impact pre-analysis is fully relational while it aggressively abstracts other semantic aspects. Second, we design a selective flow-sensitive analysis that applies flow-sensitivity only to certain program points where the main analysis is likely to need flow-sensitive information to answer given queries. The impact pre-analysis in this case is fully flow-sensitive while it aggressively abstracts other semantic aspects. The experiments show that our selective relational analysis and selective flow-sensitive analysis achieve competitive cost-precision tradeoffs when applied to real-world benchmark programs.

Contributions. In this paper, which is an extension of [Oh et al. 2014], we make the following contributions:

— We present a method for performing selective context-sensitive analysis that receives guidance from an impact pre-analysis. We provide a design of selective context-sensitive analyses (Section 4) and guidelines for designing and using an impact pre-analysis (Section 5), and show that our method ensures the impact realization (Proposition 5.17).

[1]Developing a variant of our method to the so called functional approach in [Sharir and Pnueli 1981] is a challenging and interesting future research direction.

— We show that the general idea behind our selective method is not limited to context-sensitivity. We show that our method can be used for performing relational analysis and flow-sensitive analysis in a selective way (Section 6).
— We experimentally show the effectiveness of selective analyses designed according to our method, with real-world C programs (Section 7).

Compared to the previous version [Oh et al. 2014], the present paper provides further details of our selective context-sensitive analysis (Section 4 and 5), formally proves the correctness of our method (Appendix A), and shows that the general approach behind our method is also applicable to developing a selective flow-sensitive analysis (Section 6.2).

Organization. The rest of the paper is organized as follows. Section 2 gives an informal overview of our approach. Section 3 presents the program representation that we consider. Section 4 defines a class of interprocedural program analyses that is parametrized by context-sensitivity. Section 5 shows how to design and use impact pre-analysis for finding a context-sensitivity parameter. Section 6 and Section 7 apply the general methodology behind our method into relational analysis and flow-sensitive analysis, respectively. Section 8 presents the experimental results. Section 9 discusses the related work and Section 10 concludes.

2. INFORMAL DESCRIPTION

We illustrate our approach using the interval domain and the program in Figure 1, which is adopted from `make-3.76.1`.

Example Program. Procedure `xmalloc` is a wrapper of the `malloc` function. Procedure `multi_glob` calls `xmalloc` twice, once with argument `size` (line 4) and again with an external input (line 6). The `main` procedure of this program calls procedure `f` and `g`. Procedure `f` and `g` call `multi_glob` with different argument values, which models a pattern in `make-3.76.1` that `multi_glob` often receives constant arguments.

The program contains two queries. The first query at line 5 asks whether `p` points to a buffer of size larger than 1. The other query at line 7 is similar, but this time for pointer `q`. Note that the first query always holds, but the second query is not necessarily true, because the return value of `input()` at line 6 is unknown and may well be a value not greater than 1.

Context-insensitive analysis. If we analyze the program using a context-insensitive interval analysis, we cannot prove the first query. Since the analysis is insensitive to calling contexts, it estimates the effect of `xmalloc` under all the possible inputs, and uses this same estimation as the result of every call. Note that an input to `xmalloc` at line 6 can be any integer because an unknown value is passed to `xmalloc` at line 6. As a result, the analysis concludes that `xmalloc` allocates a buffer of a size in $[-\infty, +\infty]$, and that the size of buffers allocated at line 4 and 6 (pointed to by `p` and `q`, respectively) is unknown, i.e., in $[-\infty, +\infty]$. This estimation is not strong enough to prove the first query.

Uniform context-sensitivity. A natural way to fix this precision issue is to increase the context-sensitivity. One popular approach is the k-CFA analysis [Sharir and Pnueli 1981; Shivers 1991]. It uses sequences of call sites up to length k to distinguish calling contexts of a procedure, and analyzes the procedure separately for such distinguished calling contexts. For instance, a 3-CFA analysis analyzes `xmalloc` separately for each of the following calling contexts:

```
1  char* xmalloc (int n) { return malloc(n); }
2
3  void multi_glob (int size) {
4    p = xmalloc (size);
5    assert (sizeof(p) > 1);     // Query 1
6    q = xmalloc (input());
7    assert (sizeof(q) > 1);     // Query 2
8  }
9
10 void f (int x) { multi_glob (x); }
11 void g ()      { multi_glob (4); }
12
13 int main() {
14   f (8);
15   f (16);
16   g ();
17   g ();
18 }
```

Fig. 1: Example Program

$$\begin{array}{cccc} 4\cdot 10\cdot 14 & 4\cdot 10\cdot 15 & 4\cdot 11\cdot 16 & 4\cdot 11\cdot 17 \\ 6\cdot 10\cdot 14 & 6\cdot 10\cdot 15 & 6\cdot 11\cdot 16 & 6\cdot 11\cdot 17 \end{array} \tag{1}$$

Here $a \cdot b \cdot c$ denotes a sequence of call sites a, b and c (we use the line numbers as call sites), with a being the most recent call. For instance, $4 \cdot 10 \cdot 14$ represents a series of procedure calls, first `f` at line 14, then `multi_glob` at line 10 and finally `xmalloc` at line 4. Note that in the concrete semantics, the argument `n` of `xmalloc` receives constant values 8, 16, 4, and 4, respectively, under the first four contexts above ($4\cdot10\cdot14$, $4\cdot10\cdot15$, $4\cdot11\cdot16$, and $4\cdot11\cdot17$). The 3-CFA analysis can spot this fact — it analyzes the first four calling contexts separately, and infers that a buffer of a size greater than 1 always gets allocated under those calling contexts. As a result, the analysis can deduce that the call to `xmalloc` at line 4 always returns a buffer of a size greater than 1, a conclusion strong enough to prove the first query. The second query cannot be proved even with this context-sensitivity, because the argument `n` is unknown (represented by $[-\infty, +\infty]$ in interval analysis) under the last four calling contexts in (1).

Need of selective context-sensitivity. However, using such a "uniform" context-sensitivity is not ideal. It is often too expensive to run such an analysis with high enough k, such as $k \geq 3$ that our example needs. More importantly, for many procedure calls, increasing context-sensitivity does not help — either it does not improve the analysis results of these calls, or the increased precision is not useful for answering queries. For instance, at the second query, a k-CFA analysis of any every can conclude that `p` points to a buffer of size $[-\infty, +\infty]$. Also, analyzing `g` separately for call site 16 and 17 is unnecessary because those two calls have the same effect on queries; both of the calls end up invoking `multi_glob` with the same argument 4 at line 11.

Our selective context-sensitivity. Our selective context-sensitivity aims at analyzing procedures with only needed context-sensitivity. It analyzes a procedure separately for a calling context if doing so is likely to improve the precision of the analysis and reduce false alarms in its answers for given queries. For the example program, our analysis first predicts that increasing context-sensitivity is unlikely to help answer the second query (line 7) accurately, but is likely to do so for the first query (line 5). Next, the analysis finds out that we can bring the full benefit of context-sensitivity for the first query, by distinguishing only the following four types of calling contexts of `xmalloc`:

$$\begin{array}{ll} 4 \cdot 10 \cdot 14 & 4 \cdot 10 \cdot 15 \\ 4 \cdot 11 & \text{all the other contexts} \end{array} \tag{2}$$

Note that contexts $4 \cdot 11 \cdot 16$ and $4 \cdot 11 \cdot 17$ are merged into a single context $4 \cdot 11$. This merging happens because the analysis figures out that two callers of g (line 16 and 17) do not provide any useful information for resolving the first query. Finally, the analysis analyzes the given program using the interval domain while distinguishing calling contexts above and their suffixes (i.e., $10 \cdot 14$, $10 \cdot 15$, 14, 15, 11). This selective context-sensitive analysis can prove the first query because it analyzes the call to `xmalloc` at line 4 separately for calling contexts $4 \cdot 10 \cdot 14$, $4 \cdot 10 \cdot 15$, and $4 \cdot 11$, and correctly infers that buffers of size 8, 16 and 4 get allocated, respectively.

Impact pre-analysis. Our key idea is to approximate the main analysis under full context-sensitivity using a pre-analysis, and estimate the impact of context-sensitivity on the results of the main analysis. This impact pre-analysis uses a simple (nonrelational) abstract domain and simple (disjunctive) transfer functions, and can be run efficiently even with full context-sensitivity. An abstract state of the pre-analysis represents a set of abstract states of the main analysis (rather than a set of concrete states).

For instance, we approximate the interval analysis in this example using a pre-analysis with the following abstract domain:

$$\{\bot\} \cup (\mathsf{Var} \to \{\top, \bigstar\}).$$

Here $\top$ means all intervals, and $\bigstar$ intervals of the form $[l, u]$ with $0 \leq l \leq u$. A typical abstract state in this domain is

$$[x : \top, y : \bigstar],$$

which means the following set of states in the interval domain:

$$\{[x : [l_x, u_x], y : [l_y, u_y]] \mid l_x \leq u_x \wedge 0 \leq l_y \leq u_y\}.$$

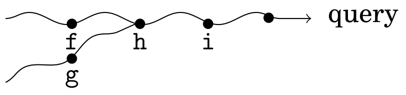
This simple abstract domain of the pre-analysis is chosen because we are interested in showing the absence of buffer overruns and the analysis proves such properties only when it finds non-negative intervals for buffer sizes and indices.

We run this pre-analysis under full context-sensitivity (i.e., ∞-CFA). For our example program, we obtain a summary of `xmalloc` with eight entries, each corresponding to a different context in (1). The third column of the table below shows this summary:

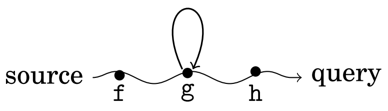
	Size of the allocated buffer in `xmalloc`	
Contexts	Main analysis	Pre-analysis
$4 \cdot 10 \cdot 14$	$[8, 8]$	$\bigstar$
$4 \cdot 10 \cdot 15$	$[16, 16]$	$\bigstar$
$4 \cdot 11 \cdot 16$	$[4, 4]$	$\bigstar$
$4 \cdot 11 \cdot 17$	$[4, 4]$	$\bigstar$
$6 \cdot 10 \cdot 14$	$[-\infty, +\infty]$	$\top$
$6 \cdot 10 \cdot 15$	$[-\infty, +\infty]$	$\top$
$6 \cdot 11 \cdot 16$	$[-\infty, +\infty]$	$\top$
$6 \cdot 11 \cdot 17$	$[-\infty, +\infty]$	$\top$

The second column of the table shows the results of the interval analysis with full context-sensitivity. Note that the pre-analysis in this case precisely estimates the impact of context-sensitivity: it identifies calling contexts (i.e., the first four contexts in the table) where the interval analysis accurately tracks the size of the allocated buffer in `xmalloc` under the full context-sensitivity. In general, our pre-analysis might lose precision and use $\top$ more often than in the ideal case. However, even when such approximation occurs, it does so only in a sound manner—if the pre-analysis computes $\bigstar$ for the size of a buffer, the interval analysis under full context-sensitivity is guaranteed to compute a non-negative interval.

Use of pre-analysis results. From the pre-analysis results, we select calling contexts that help improve the precision regarding given queries. We first identify queries whose expressions are assigned with $\bigstar$ in the pre-analysis run. In our example, the pre-analysis assigns $\bigstar$ to expression `sizeof(p)` in the first query. We regard this as a good indication that the interval analysis under full context-sensitivity is likely to estimate the value of `sizeof(p)` accurately. Then, for each query that is judged promising, we consider the slice of the program that contributes to the query. We conclude that all the calls made in the slice should be tracked precisely. For example, if a slice for a query looks as follows:



Then, we derive calling contexts `f`, `g`, $\{\mathtt{h}\cdot\mathtt{f},\mathtt{h}\cdot\mathtt{g}\}$, and $\{\mathtt{i}\cdot\mathtt{h}\cdot\mathtt{f},\mathtt{i}\cdot\mathtt{h}\cdot\mathtt{g}\}$ for `f`, `g`, `h`, and `i`, respectively. However, if a slice involves a recursive call as follows:



we exclude the query since otherwise, we need infinitely many different calling contexts. In our example program, the slice for the first query includes all the execution paths from lines 11, 14, and 15 to line 5. Note that call-sites 16 and 17 are not included in the slice, since procedure `g` is called with no actual arguments and does not make dependency on the queries of interest. Thus, all the calling contexts of `xmalloc` in this slice are:

$$4\cdot 10\cdot 14 \qquad 4\cdot 10\cdot 15 \qquad 4\cdot 11 \tag{3}$$

Our analysis decides to distinguish these contexts and their suffixes. (We distinguish all of these contexts to work against the precision degradation caused by widening. For instance, in our example program, it is not sufficient to simply distinguish the call sites to `xmalloc` (line 4 and 6). Suppose we adopt this simple strategy for selective context-sensitivity, and do not separate the calling contexts of `multi_glob` and `f`. Because of the spurious interprocedural cycles [Oh and Yi 2010] caused by the joined calling contexts, the analysis should apply the widening operation when analyzing `multi_glob` and `f`. At the first call site (line 14), `f` is analyzed with the parameter value $[8,8]$, and at the second call site (line 15), `f` is analyzed with $[16,16]$. At this point, the analysis combines these parameter values with the widening operator, and sets the estimation of the parameter `x` to the interval $[8,+\infty]$. As `f` calls `multi_glob`, the parameter `size` of `multi_glob` has the interval $[8,+\infty]$. This estimation is weakened again at the call of `multi_glob` at line 11, and it becomes $[8,+\infty]\nabla[4,4]=[-\infty,+\infty]$. Hence, the analysis fails to prove that the call at line 4 returns a buffer of size >1. This is why our selectively context-sensitive analysis analyzes `xmalloc` for the three calling contexts in (2) separately.)

Impact realization. Our method guarantees that the impact estimation under full context-sensitivity pays off at the subsequent selective context-sensitive analysis. That is, in our example program, the selective main analysis, which distinguishes only the contexts in (2), is guaranteed to assign nonnegative intervals to `sizeof(p)` at the first query. This guarantee holds because our selective context-sensitive analysis distinguishes all the calling contexts that matter for the selected queries (Section 5.2) and ensures that undistinguished contexts are isolated from the distinguished contexts (Section 4). For instance, even if the call to `xmalloc` at line 6 is analyzed in a context-insensitive way, the result does not taint the precise summary of the other call to `xmalloc` at line 4.

The impact realization property provides a minimum guarantee and increases the predictability of our method. Suppose that the impact realization does not hold. Then, the pre-analysis might possibly give $\bigstar$ for a query for which the main analysis would produce $\top$. In that case, our method would still select that query even though the main analysis under full context-sensitivity would end up failing to prove the query's safety. The impact realization is therefore important in that it guarantees such cases do not occur.

Limitation. Our approach has weakness in two ways; it might choose to context-sensitively analyze trivial queries requiring no context-sensitivity, or miss the ones that can be actually proven under a proper context-sensitivity. A nontrivial result (i.e., $\bigstar$) for a query could have been from both intraprocedural and interprocedural value dependencies and the pre-analysis itself does not distinguish between the two. If there have been only intraprocedural dependencies that contribute to a nontrivial result, the main-analysis would anyway apply context-sensitivity for the query even though it is unnecessary. Also, the pre-analysis could have given a trivial result (i.e., $\top$) due to its own limitation in the value abstraction, not because the main-analysis could not prove the query. In this case, our approach fails to catch the opportunity to use context-sensitivity for improving the precision.

However, we can easily filter out trivial queries that do not involve any procedure calls. For instance, when analyzing the following snippet, we could easily check that the query in function `foo` has no interprocedural value dependencies and therefore can be pruned out.

```
int foo() {
  int a[4];
  int x = 1;
  a[0] = 1;  // buffer access (query)
}
```

Furthermore, we observed that a trivial result for a query often means it is too difficult even for the abstract domain of the main analysis to handle, regardless of the degree of context-sensitivity used. Such cases include bitwise operations which are vastly approximated by the numerical abstract domain of our main analysis, and external function calls which are difficult to analyze precisely on their own.

Application to other selective analyses. Behind our approach lies a general principle for developing a static analysis that selectively uses precision-improving techniques, such as context-sensitivity, relational abstract domains, and flow-sensitivity. The principle is to run an impact pre-analysis that finds out when and where the main static analysis under its best precision setting is likely to have an accurate result, and to choose appropriate analysis parameters of the main analysis based on the pre-analysis results.

For instance, consider the octagon analysis [Miné 2006], which tracks constraints of form $\pm x \pm y \leq c$ (where $c \in \mathbb{Z} \cup \{+\infty\}$) for a pair of variable x and y. Since tracking constraints of all possible variable pairs is inefficient and often intractable, we want to select only some of them that might be tracked precisely by the octagon domain and also helpful to prove the queries of interest. In order to achieve this goal, we design an impact pre-analysis that aims at finding when and where the original octagon analysis is likely to infer precise relationships between program variables.

More specifically, this pre-analysis tracks constraints of form $\pm x \pm y \leq a$ between all possible pairs of x and y, but it uses a simple domain of only two abstract values $\top$ and $\bigstar$ as bound a, instead of all integers and ∞ as the octagon analysis does. Here $x+y \leq \top$ represents all octagon constraints of form $x+y \leq c$ including the case when $c = +\infty$, whereas $x + y \leq \bigstar$ just excludes that case. The observation behind the design of the abstract domain is that the octagon analysis could often prove the safety of a buffer access when it had inferred a constraint of an integer bound as the invariant. Based on the results of this pre-analysis, we can estimate the octagonal constraints that help improve the precision regarding given queries, and guide our selective octagon analysis to track only these constraints.

In Section 6.1, we describe this selective octagon analysis in details. In section 6.2 we describe another application of our approach to flow-sensitivity.

3. PROGRAM REPRESENTATION

Control Flow Graph. We assume that a program $\mathcal{P}$ is represented by a control flow graph $(\mathbb{C}, \rightarrow, \mathbb{F}, \iota)$ where $\mathbb{C}$ is the finite set of nodes, $(\rightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ denotes the control flow relation between nodes, $\mathbb{F}$ is the set of procedure ids, and $\iota \in \mathbb{C}$ is the entry node of the `main` procedure. The entry node ι does not have predecessors. A node $c \in \mathbb{C}$ in the program is one of the five types:

$$\begin{array}{rclr} \mathbb{C} & = & \mathbb{C}_e & \text{(Entry Nodes)} \\ & \uplus & \mathbb{C}_x & \text{(Exit Nodes)} \\ & \uplus & \mathbb{C}_c & \text{(Call Nodes)} \\ & \uplus & \mathbb{C}_r & \text{(Return Nodes)} \\ & \uplus & \mathbb{C}_i & \text{(Internal Nodes)} \end{array}$$

Each procedure $f \in \mathbb{F}$ has one entry node and one exit node. Given a node $c \in \mathbb{C}$, $\mathsf{fid}(c)$ denotes the procedure enclosing the node. Each call-site in the program is represented by a pair of call and return nodes. Given a return node $c \in \mathbb{C}_r$, we write $\mathsf{callof}(c)$ for the corresponding call node. We denote the set of call edges by $\rightarrowtail$:

$$(\rightarrowtail) = \{(c_1, c_2) \mid c_1 \rightarrow c_2 \ \wedge \ c_1 \in \mathbb{C}_c \ \wedge \ c_2 \in \mathbb{C}_e\}$$

and the set of return edges by $\dashrightarrow$:

$$(\dashrightarrow) = \{(c_1, c_2) \mid c_1 \rightarrow c_2 \ \wedge \ c_1 \in \mathbb{C}_x \ \wedge \ c_2 \in \mathbb{C}_r\}.$$

We assume for simplicity that there are no indirect function calls such as calls via function pointers, that is, the call and return edges respect the following conditions:

$$\forall c_1, c_2, c_3 \in \mathbb{C}. \begin{cases} c_1 \rightarrowtail c_2 \wedge c_1 \rightarrowtail c_3 & \Longrightarrow \ c_2 = c_3 \\ c_2 \dashrightarrow c_1 \wedge c_3 \dashrightarrow c_1 & \Longrightarrow \ c_2 = c_3 \end{cases}$$

Primitive Command. We associate a primitive command with each node c of our control flow graph, and denote it by $\mathsf{cmd}(c)$. The set of possible primitive commands is specified by the following grammar:

$$cmd \ \rightarrow \ skip \mid x := e$$

where e is an arithmetic expression:

$$e \to n \mid x \mid e + e \mid e - e$$

We denote the set of all program variables by Var.

For simplicity, we handle parameter passing and return values of procedures via simple syntactic encoding.[2] Recall that we represent a call statement $x := f_p(e)$ (where p is a formal parameter of procedure f) with call and return nodes. In our program, the call node has command $p := e$, so that the actual parameter e is assigned to the formal parameter p. For return values, we assume that each procedure f has a variable r_f and the return value is assigned to r_f: that is, we represent return statement `return` e of procedure f by $r_f := e$. The return node has command $x := r_f$, so that the return value is assigned to the original return variable. We assume that there are no global variables in the program, all parameters and local variables of procedures are distinct, and there are no recursive procedures.

4. SELECTIVE CONTEXT-SENSITIVE ANALYSIS WITH PARAMETER K

We consider selective context-sensitive analyses specified by the following data:

(1) A domain $\mathbb{S}$ of abstract states. We assume that this domain has a complete lattice structure:

$$(\mathbb{S}, \sqsubseteq, \bot, \top, \sqcup, \sqcap).$$

(2) An initial abstract state at the entry of the `main` procedure:

$$s_I \in \mathbb{S}.$$

(3) An abstract semantics of every primitive command cmd:

$$[\![cmd]\!] : \mathbb{S} \to \mathbb{S}.$$

We require that semantic function $[\![cmd]\!]$ be monotone.

(4) A context selector K that maps procedures to sets of calling contexts (sequences of call nodes):

$$K \in \mathbb{F} \to \wp(\mathbb{C}_c^*)$$

For procedure f, the set $K(f)$ specifies calling contexts that the analysis should differentiate while analyzing the procedure. We sometimes abuse the notation and denote by K the entire set of calling contexts in K: we write $\kappa \in K$ to denote that $\kappa \in \bigcup_{f \in \mathbb{F}} K(f)$.

With the above data, we design a selective context-sensitive analysis as follows. First, we differentiate nodes with contexts in K, and define a set $\mathbb{C}_K \subseteq \mathbb{C} \times \mathbb{C}_c^*$ of context-enriched nodes:

$$\mathbb{C}_K = \{(c, \kappa) \mid c \in \mathbb{C} \ \wedge \ \kappa \in K(\mathsf{fid}(c))\}.$$

The control flow relation $(\to) \subseteq \mathbb{C} \times \mathbb{C}$ is extended to $\to_K$ on $\mathbb{C}_K$:

Definition 4.1 $(\to_K)$. $(\to_K) \subseteq \mathbb{C}_K \times \mathbb{C}_K$ is the context-enriched control flow relation:

$$(c, \kappa) \to_K (c', \kappa') \ \textit{iff}$$

$$\begin{cases} c \to c' \ \wedge \ \kappa' = \kappa & (c' \notin \mathbb{C}_e \uplus \mathbb{C}_r) \\ c \to c' \ \wedge \ \kappa' = c ::_K \kappa & (c \in \mathbb{C}_c \ \wedge \ c' \in \mathbb{C}_e) \\ c \to c' \ \wedge \ \kappa = \mathsf{callof}(c') ::_K \kappa' & (c \in \mathbb{C}_x \ \wedge \ c' \in \mathbb{C}_r) \end{cases}$$

[2]For presentation brevity. Alternatively, we can handle parameter passing and return values directly in the (abstract) semantics, which our implementation in Section 7 follows.

where $(::_K) \in \mathbb{C}_c \times \mathbb{C}_c^* \rightarrow \mathbb{C}_c^*$ updates contexts according to K:

$$c ::_K \kappa = \begin{cases} c \cdot \kappa & (c \cdot \kappa \in K) \\ \epsilon & \textit{otherwise} \end{cases}$$

where ϵ is the empty call sequence.

In our analysis, ϵ is used to represent *all the other contexts* not included in K, and we assume that K includes ϵ if it is necessary. For instance, consider a program where f has three different calling contexts κ_1, κ_2, and κ_3. When the analysis differentiates κ_1 only, undistinguished contexts κ_2 and κ_3 are represented by ϵ. Thus, $K(f) = \{\kappa_1, \epsilon\}$. Note that our analysis isolates undistinguished contexts from distinguished ones: ϵ means only κ_2 or κ_3, not κ_1.

Example 4.2. The analysis is context-insensitive when $K = \lambda f.\{\epsilon\}$ and fully context-sensitive when $K = \lambda f.\mathbb{C}_c^*$. Our selective context-sensitive analysis in Section 2 uses the following context selector K:

$$\begin{aligned} \texttt{main} &\mapsto \{\epsilon\} \\ \texttt{f} &\mapsto \{14, 15\} \\ \texttt{g} &\mapsto \{\epsilon\} \\ \texttt{multi\_glob} &\mapsto \{10 \cdot 14, 10 \cdot 15, 11\} \\ \texttt{xmalloc} &\mapsto \{4 \cdot 10 \cdot 14, 4 \cdot 10 \cdot 15, 4 \cdot 11, \epsilon\} \end{aligned}$$

Procedures `f` and `multi_glob` do not have ϵ, as all of their calling contexts are prescribed in K.

Next, we define the abstract domain $\mathbb{D}$ of the analysis:

$$\mathbb{D} = (\mathbb{C}_K \rightarrow \mathbb{S}) \tag{4}$$

The analysis keeps multiple abstract states at each program node c, one for each context $\kappa \in K(\mathsf{fid}(c))$. The abstract transfer function F of the analysis works on $\mathbb{C}_K$, and it is defined as follows:

$$F(X)(c, \kappa) = [\![\mathsf{cmd}(c)]\!](\bigsqcup_{(c_0,\kappa_0) \rightarrow_K (c,\kappa)} X(c_0, \kappa_0)). \tag{5}$$

The static analysis computes an abstract element $X \in \mathbb{D}$ that over-approximates all the concrete states summarized by s_I and forms an inductive invariant of the given program:

$$s_I \sqsubseteq X(\iota, \epsilon) \ \wedge \ \forall (c, \kappa) \in \mathbb{C}_K.\ F(X)(c, \kappa) \sqsubseteq X(c, \kappa) \tag{6}$$

In general, many X can satisfy the condition in (6). Choosing one among these solutions is up to each static analysis, and depends on its fixpoint algorithm. Some analyses compute the least X satisfying (6), where abstract elements are ordered pointwise:

$$X \sqsubseteq Y \ \textit{ iff } \ \forall (c, \kappa) \in \mathbb{C}_K.\ X(c, \kappa) \sqsubseteq Y(c, \kappa).$$

Other analyses use a widening operator [Cousot and Cousot 1977; 1992], and compute not necessarily the least, but some solution of (6). We reminder the reader that a widening operator $\bigtriangledown$ is a binary function on $\mathbb{D}$:

$$\bigtriangledown : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{D}$$

such that

(1) $X \sqsubseteq X \bigtriangledown Y$ and $Y \sqsubseteq X \bigtriangledown Y$ for all $X, Y \in \mathbb{D}$; and

(2) for every sequence $\{X_n\}$ in $\mathbb{D}$, the following widened sequence $\{Y_n\}$ converges after some finite step:

$$Y_0 = X_0, \qquad Y_{n+1} = Y_n \bigtriangledown X_{n+1}.$$

These analyses compute the limit of the following converging sequence:

$$\begin{array}{rcl} X_0 & = & \lambda(c,\kappa).\, \text{if } ((c,\kappa) = (\iota,\epsilon)) \text{ then } s_I \text{ else } \bot, \\ X_{n+1} & = & X_n \bigtriangledown F(X_n). \end{array}$$

Example 4.3. The interval analysis is a standard example that uses a widening operator. Let $\mathbb{I}$ be the domain of intervals defined by

$$\mathbb{I} = \{[l,u] \mid l,u \in \mathbb{Z} \cup \{-\infty,+\infty\} \ \wedge\ l \le u\}$$

and order elements in this domain as follows:

$$[l,u] \sqsubseteq [l',u'] \quad \textit{iff} \quad l' \le l \ \wedge\ u \le u'.$$

Using this domain, we specify the rest of the analysis:

(1) The abstract states are $\bot$ or functions from program variables (Var) to their interval values ($\mathbb{I}$):

$$\mathbb{S} = \{\bot\} \cup (\mathsf{Var} \to \mathbb{I})$$

An abstract state $s \in \mathbb{S}$ specifies upper and lower bounds for every program variable, and it means concrete states satisfying these constraints. Abstract states are ordered pointwise, and form a complete lattice:

$$s \sqsubseteq s' \quad \textit{iff} \quad (s = \bot) \vee (\forall x \in \mathsf{Var}.\, s(x) \sqsubseteq s'(x))$$

(2) The initial abstract state is:

$$s_I(x) = [-\infty, +\infty].$$

(3) The abstract semantics of primitive commands is:

$$\begin{array}{rcl} [\![\mathit{skip}]\!](s) & = & s \\ [\![x := e]\!](s) & = & \begin{cases} s[x \mapsto [\![e]\!](s)] & (s \ne \bot) \\ \bot & (s = \bot) \end{cases} \end{array}$$

where $[\![e]\!]$ is the abstract evaluation of the expression e with interval values: for $s \in \mathbb{S}$ with $s \ne \bot$,

$$\begin{array}{rcl} [\![n]\!](s) & = & [n,n] \\ [\![x]\!](s) & = & s(x) \\ [\![e_1 + e_2]\!](s) & = & [\![e_1]\!](s) + [\![e_2]\!](s) \\ [\![e_1 - e_2]\!](s) & = & [\![e_1]\!](s) - [\![e_2]\!](s). \end{array}$$

In the evaluation, we use standard operators for adding and subtracting interval values, which we recall below:

$$\begin{array}{rcl} [a,b] + [c,d] & = & [a+c, b+d], \\ [a,b] - [c,d] & = & [a-d, b-c] \end{array}$$

(4) The last component of the analysis is a widening operator. It is based on the following operators of type $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$:

$$\begin{array}{rcl} [l,u] \sqcup [l',u'] & = & [\min(l,l'), \max(u,u')] \\ [l,u] \bigtriangledown_I [l',u'] & = & [\textit{if } (l' < l) \textit{ then} \\ & & \quad (\text{if } (l' < 0) \text{ then } -\infty \text{ else } 0) \text{ else } l, \\ & & \ \textit{if } (u' > u) \textit{ then } +\infty \textit{ else } u] \end{array}$$

Note that the above widening operator uses 0 as a threshold, which is useful when used for proving buffer-overrun safety. The analysis carefully chooses a set of nodes $\mathbb{C}_w$, such as all the loop headers, function entries and exits, and uses the following widening operator:

$$\begin{array}{l}(X \mathbin{\nabla} Y)(c,\kappa) = \\ \quad \text{if } (X(c,\kappa) = \bot \lor Y(c,\kappa) = \bot \lor c \notin \mathbb{C}_w) \\ \quad\quad \text{then } (X(c,\kappa) \sqcup Y(c,\kappa)) \\ \quad\quad \text{else } (\lambda x.\, X(c,\kappa)(x) \mathbin{\nabla_I} Y(c,\kappa)(x)).\end{array}$$

Queries. Queries are triples in $\mathcal{Q} \subseteq \mathbb{C} \times \mathbb{S} \times \mathsf{Var}$, and they are given as an input to our static analysis. A query (c, s, x) represents an assertion that every reachable concrete state at node c is over-approximated by the abstract state s. The last component x describes that the query is concerned with the value of variable x. For instance, in the interval analysis, a typical query is

$$(c,\ \ \lambda y.\, \text{if } (y = x) \text{ then } [0,\infty] \text{ else } \top,\ \ x)$$

for some variable x. It asserts that at program node c, the variable x should always have a non-negative value. Proving the queries or identifying those that are likely to be violated is the goal of the analysis.

5. IMPACT PRE-ANALYSIS FOR FINDING K

Suppose that we would like to develop a selective context-sensitive analysis in Section 4 for a given program and given queries, using one of the existing abstract domains specified by the following data:

$$(\mathbb{S},\ \ s_I \in \mathbb{S},\ \ [\![-]\!] : \mathbb{S} \to \mathbb{S}),$$

where $\mathbb{S}$ is the domain of abstract states, s_I the initial abstract state, and $[\![cmd]\!] : \mathbb{S} \to \mathbb{S}$ the abstract semantics of a primitive command cmd. To achieve our aim, we need to construct K, a specification on context-sensitivity for the given program and queries. Once this construction is done, the rest is standard. The analysis can analyze the program under the partial context-sensitivity, using an induced abstract domain $\mathbb{D}$ and transfer function $F : \mathbb{D} \to \mathbb{D}$ from equation (4) and (5). We assume that the analysis employs the fixpoint algorithm based on widening operation $\nabla : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$.

How could we automatically choose an effective context selector K that balances the precision and cost of the induced interprocedural analysis? In this section, we give an answer to this question. In Section 5.1, we present an impact pre-analysis which estimates the behavior of the main analysis $(\mathbb{S}, s_I, [\![-]\!])$ under full context-sensitivity. In Section 5.2, we describe how we could use the results of the pre-analysis for constructing K. Throughout the section, we fix our main analysis to $(\mathbb{S}, s_I, [\![-]\!])$.

5.1. Designing an Impact Pre-Analysis

An impact pre-analysis for context-sensitivity aims at estimating the main analysis $(\mathbb{S}, s_I, [\![-]\!])$ under full context-sensitivity. It is specified by the following data:

$$(\mathbb{S}^\sharp,\ \ s_I^\sharp \in \mathbb{S}^\sharp,\ \ [\![-]\!]^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp,\ \ K_\infty).$$

This specification and the way that the data are used in our pre-analysis are fairly standard. $\mathbb{S}^\sharp$ and $[\![cmd]\!]^\sharp$ are, respectively, the domain of abstract states and the abstract semantics of cmd used by the pre-analysis, and $s_I^\sharp$ is an initial state. $K_\infty = \lambda f.\mathbb{C}_c^*$ is the context selector for full context-sensitivity. The pre-analysis uses the abstract domain

$$\mathbb{D}^\sharp = \mathbb{C}_{K_\infty} \to \mathbb{S}^\sharp$$

and the following transfer function $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ for the given program:

$$F^\sharp(X)(c,\kappa) = [\![\mathsf{cmd}(c)]\!]^\sharp(\bigsqcup_{(c_0,\kappa_0)\to_{K_\infty}(c,\kappa)} X(c_0,\kappa_0)).$$

It computes the least X satisfying

$$s_I^\sharp \sqsubseteq X(\iota,\epsilon) \ \wedge\ \forall (c,\kappa) \in \mathbb{C}_K.\ F^\sharp(X)(c,\kappa) \sqsubseteq X(c,\kappa) \tag{7}$$

What is less standard is the soundness and computability conditions for our pre-analysis, which provide a guideline on the design of the pre-analysis. Let us discuss these conditions separately.

Soundness condition. Intuitively, our soundness condition says that all the components of the pre-analysis have to over-approximate the corresponding ones of the main analysis.[3] This is identical to the standard soundness requirement of a static program analysis, except that the condition is stated not over the concrete semantics of a given program, but over the main analysis. Also, the soundness is necessary to establish later the *impact realization* property of the designed pre-analysis (PROPOSITION 6.3).
The condition has the following four requirements:

(1) There should be a concretization function γ from $\mathbb{S}^\sharp$ to $\wp(\mathbb{S})$
$$\gamma : \mathbb{S}^\sharp \to \wp(\mathbb{S}).$$
This function formalizes the fact that an abstract state of the pre-analysis means a set of abstract states of the main analysis.
(2) The initial abstract state of the pre-analysis has to overapproximate the initial state of the main analysis, i.e.,
$$s_I \in \gamma(s_I^\sharp).$$
(3) The abstract semantics of primitive commands in the pre-analysis should be sound with respect to that of the main analysis:
$$\forall s \in \mathbb{S}, s^\sharp \in \mathbb{S}^\sharp.\ s \in \gamma(s^\sharp) \implies [\![cmd]\!](s) \in \gamma([\![cmd]\!]^\sharp(s^\sharp)).$$
(4) The join operation of the pre-analysis's abstract domain over-approximates the widening operation of the main analysis. The join operation $\sqcup$ on $\mathbb{D}^\sharp = \mathbb{C}_{K_\infty} \to \mathbb{S}^\sharp$ is defined as follows:
$$(X^\sharp \sqcup Y^\sharp)(c,\kappa) = X^\sharp(c,\kappa) \sqcup Y^\sharp(c,\kappa).$$
This join operation should approximate the widening operation of the main analysis: for all $X, Y \in \mathbb{D}$ and $X^\sharp, Y^\sharp \in \mathbb{D}^\sharp$,
$$(X \in \gamma(X^\sharp) \wedge Y \in \gamma(Y^\sharp)) \implies X \bigtriangledown Y \in \gamma(X^\sharp \sqcup Y^\sharp).$$

The purpose of our condition is that the impact pre-analysis over-approximates the fully context-sensitive main analysis:

LEMMA 5.1. *Let $M \in \mathbb{D}$ be the main analysis result, i.e., a solution of* (6) *under full context-sensitivity (K=K_∞). Let $P \in \mathbb{D}^\sharp$ be the pre-analysis result, i.e., the least solution of* (7). *Then, the pre-analysis result over-approximates the main analysis result:*

$$\forall c \in \mathbb{C}, \kappa \in \mathbb{C}_c^*.\ M(c,\kappa) \in \gamma(P(c,\kappa)).$$

PROOF. Immediate from the abstract interpretation framework [Cousot and Cousot 1977; 1992]. □

[3]We design a pre-analysis as an over-approximation of the main analysis, because an under-approximating pre-analysis would be too optimistic in context selection (because its analysis result contains more ★ than the over-approximated pre-analysis) and the resulting selective main analysis is hardly cost-effective.

Computability condition. The next condition is for the computability of our pre-analysis. The pre-analysis we are designing over-approximates a main analysis of the best possible precision which is in most cases incomputable or intractable. This computability condition ensures that the pre-analysis, though estimating the most precise main analysis, remains to be computable. Furthermore, the condition yields a pre-analysis that can be computed by an efficient algorithm since it requires the analysis to be of a simple form. The condition consists of two requirements:

(1) The abstract states are $\bot$ or functions from program variables to abstract values:
$$\mathbb{S}^\sharp = \{\bot\} \cup (\mathsf{Var} \to \mathbb{V})$$
where $\mathbb{V}$ is a finite complete lattice
$$(\mathbb{V}, \sqsubseteq_v, \bot_v, \top_v, \sqcup_v, \sqcap_v).$$
We extend this lattice structure point-wise and equip $\mathbb{S}^\sharp$ with usual lattice operators. The definition of lattice operators are as follows:
$$\begin{array}{rcl} s \sqsubseteq s' & \textit{iff} & (s = \bot) \vee (\forall x.\, s(x) \sqsubseteq_v s'(x)) \\ \top & = & \lambda x.\, \top_v \\ \bot \sqcup s' & = & s' \sqcup \bot \quad = \quad s' \\ s \sqcup s' & = & \lambda x.\, s(x) \sqcup_v s'(x) \qquad (s \neq \bot, s' \neq \bot) \\ \bot \sqcap s' & = & s' \sqcap \bot \quad = \quad \bot \\ s \sqcap s' & = & \lambda x.\, s(x) \sqcap_v s'(x) \qquad (s \neq \bot, s' \neq \bot) \end{array}$$
An initial abstract state:
$$s^\sharp_I = \lambda x.\, \top_v.$$

(2) The abstract semantics of primitive commands has a simple form involving only join operation and constant abstract value, which is defined as follows:
$$\begin{array}{rcl} [\![\mathit{skip}]\!]^\sharp(s) & = & s \\ [\![x := e]\!]^\sharp(s) & = & \begin{cases} s[x \mapsto [\![e]\!]^\sharp(s)] & (s \neq \bot) \\ \bot & (s = \bot) \end{cases} \end{array}$$
where $[\![e]\!]^\sharp$ has the following form: for every $s \neq \bot$,
$$[\![e]\!]^\sharp(s) = s(x_1) \sqcup \ldots \sqcup s(x_n) \sqcup v$$
for some variables $x_1, \ldots, x_n$ and an abstract value $v \in \mathbb{V}$, all of which are fixed for the given e. We denote these variables and the value by
$$\mathsf{var}(e) = \{x_1, \ldots, x_n\}, \qquad \mathsf{const}(e) = v.$$
The exact choice of $x_1, \ldots, x_n$ and v depends on each analysis.

Example 5.2 (*Impact Pre-Analysis for the Interval Analysis*). We design a pre-analysis for our interval analysis in Example 4.3, which satisfies our soundness and computability conditions. The pre-analysis aims at predicting which variables get associated with non-negative intervals when the program is analyzed by an interval analysis with full context-sensitivity K_∞.

(1) Let $\mathbb{V} = \{\bot_v, \bigstar, \top_v\}$ be a lattice such that $\bot_v \sqsubseteq_v \bigstar \sqsubseteq_v \top_v$. Define a function $\gamma_v : \{\bot_v, \bigstar, \top_v\} \to \wp(\mathbb{I})$ as follows:
$$\begin{array}{rcl} \gamma_v(\top_v) & = & \mathbb{I} \\ \gamma_v(\bigstar) & = & \{[a,b] \in \mathbb{I} \mid 0 \leq a\} \\ \gamma_v(\bot_v) & = & \emptyset \end{array}$$
This function determines the meaning of each element in $\mathbb{V}$ in terms of a collection of intervals. The only non-trivial case is $\bigstar$ which denotes all non-negative inter-

vals. We include this case because the main analysis should infer non-negative intervals on buffer accesses in order to prove buffer-overrun safety.

(2) Abstract states are $\bot$ or functions from program variables (Var) to their flow values ($\mathbb{V}$):

$$\mathbb{S}^\sharp = \{\bot\} \cup (\mathsf{Var} \to \mathbb{V}).$$

The meaning of abstract states in $\mathbb{S}^\sharp$ is given by a concretization function γ. Let $\mathbb{S}$ be the abstract domain of the interval analysis, and s a non-$\bot$ element of $\mathbb{S}^\sharp$.

$$\begin{aligned} \gamma(\bot) &= \{\bot\} \\ \gamma(s) &= \{s \in \mathbb{S} \mid s = \bot \vee \forall x \in \mathsf{Var}.\ s(x) \in \gamma_v(s(x))\}. \end{aligned}$$

(3) Initial abstract state:

$$s_I^\sharp = \top = \lambda x.\top_v.$$

(4) Abstract semantics of primitive commands:

$$\begin{array}{rcl} [\![skip]\!]^\sharp(s) &=& s \\ [\![x := e]\!]^\sharp(s) &=& \left\{ \begin{array}{ll} s[x \mapsto [\![e]\!]^\sharp(s)] & (s \neq \bot) \\ \bot & (s = \bot) \end{array} \right. \end{array}$$

where $[\![e]\!]^\sharp$ is defined as follows: for every $s \neq \bot$,

$$\begin{array}{rcl} [\![n]\!]^\sharp(s) &=& \text{if } (n \geq 0) \text{ then } \bigstar \text{ else } \top_v \\ [\![x]\!]^\sharp(s) &=& s(x) \\ [\![e_1 + e_2]\!]^\sharp(s) &=& [\![e_1]\!]^\sharp(s) \sqcup_v [\![e_2]\!]^\sharp(s) \\ [\![e_1 - e_2]\!]^\sharp(s) &=& \top_v \end{array}$$

The analysis approximately tracks numbers, but distinguishes the non-negative cases from general ones: non-negative numbers get abstracted to $\bigstar$ by the analysis, but negative numbers are represented by $\top_v$. Observe that the $+$ operator is interpreted as the least upper bound $\sqcup_v$, so that $e_1 + e_2$ evaluates to $\bigstar$ only when both e_1 and e_2 evaluates to $\bigstar$. This implements the intuitive fact that the addition of two non-negative intervals gives another non-negative interval. For expressions involving subtractions, the analysis simply produces $\top_v$. The above pre-analysis satisfies the soundness and computability conditions of our impact pre-analysis. That is, $[\![e]\!]^\sharp$ has the form of

$$[\![e]\!](s) = s(x_1) \sqcup_v \ldots \sqcup_v s(x_n) \sqcup_v v$$

for all expressions. Also, it is easy to check that this abstract semantics of our pre-analysis is sound with respect to the abstract semantics of the interval analysis in Example 4.3. Note that the join operation of our pre-analysis over-approximates the zero-threshold widening operator in Example 4.3.

Running the pre-analysis via reachability-based algorithm. The class of our pre-analyses enjoys efficient algorithms (e.g., [Reps et al. 1995; Deutsch 1997]) for computing the least solution X that satisfies (7), even though it is fully context-sensitive. For our purpose, we provide a variant of the graph reachability-based algorithm in [Reps et al. 1995]. Next, we go through each step of our algorithm while introducing concepts necessary to understand it. In the rest of this section, we interchangeably write K for K_∞.

First, our algorithm constructs the *value-flow graph* of the given program, which is a finite graph $(\Theta, \hookrightarrow)$ defined as follows:

$$\Theta = \mathbb{C} \times \mathsf{Var}, \qquad (\hookrightarrow) \subseteq \Theta \times \Theta$$

Set Θ of vertices consists of pairs of program nodes and variables, and relation $(\hookrightarrow)$ describes edges between vertices.

Definition 5.3 ($\hookrightarrow$). The value-flow relation $(\hookrightarrow) \subseteq (\mathbb{C} \times \mathsf{Var}) \times (\mathbb{C} \times \mathsf{Var})$ links the vertices in Θ based on how the value of one variable flows to another at each primitive command:

$$(c, x) \hookrightarrow (c', x') \ \mathit{iff} \ \begin{cases} c \to c' \ \wedge \ x = x' & (\mathsf{cmd}(c') = \mathit{skip}) \\ c \to c' \ \wedge \ x = x' & (\mathsf{cmd}(c') = y := e \ \wedge \ y \neq x') \\ c \to c' \ \wedge \ x \in \mathsf{var}(e) & (\mathsf{cmd}(c') = y := e \ \wedge \ y = x') \end{cases}$$

We can extend relation $\hookrightarrow$ to its context-enriched version $\hookrightarrow_K$ as follows:

Definition 5.4 ($\hookrightarrow_K$). The context-enriched value-flow relation $(\hookrightarrow_K) \subseteq (\mathbb{C}_K \times \mathsf{Var}) \times (\mathbb{C}_K \times \mathsf{Var})$ links the vertices in $\mathbb{C}_K \times \mathsf{Var}$ according to the specification below:

$$((c, \kappa), x) \hookrightarrow_K ((c', \kappa'), x') \ \mathit{iff} \ \begin{cases} (c, \kappa) \to_K (c', \kappa') \ \wedge \ x = x' & (\mathsf{cmd}(c') = \mathit{skip}) \\ (c, \kappa) \to_K (c', \kappa') \ \wedge \ x = x' & (y \neq x') \\ (c, \kappa) \to_K (c', \kappa') \ \wedge \ x \in \mathsf{var}(e) & (y = x') \end{cases}$$

(where $\mathsf{cmd}(c')$ in the last two cases is $y := e$)

Second, the algorithm computes the interprocedurally-valid reachability relation $(\hookrightarrow_K^\dagger) \subseteq \Theta \times \Theta$:

Definition 5.5 ($\hookrightarrow_K^\dagger$). The reachability relation $(\hookrightarrow_K^\dagger) \subseteq \Theta \times \Theta$ connects two vertices when one node can reach the other via an interprocedurally-valid path:

$$(c, x) \hookrightarrow_K^\dagger (c', x') \ \mathit{iff} \ \exists \kappa, \kappa'. \, (\iota, \epsilon) \to_K^* (c, \kappa) \ \wedge \ ((c, \kappa), x) \hookrightarrow_K^* ((c', \kappa'), x').$$

We use the tabulation algorithm in [Reps et al. 1995] for computing $(\hookrightarrow_K^\dagger)$. While computing $(\hookrightarrow_K^\dagger)$, the algorithm also collects the set C of reachable nodes:

$$C = \{c \mid \exists \kappa. \, (\iota, \epsilon) \to_K^* (c, \kappa)\}. \tag{8}$$

Third, our algorithm computes a set Θ_v of generators for each abstract value v in $\mathbb{V}$. Generators for v are vertices in Θ whose commands join v in their abstract semantics:

$$\begin{aligned} \Theta_v = \ & \{(c, x) \mid \mathsf{cmd}(c) = x := e \wedge \mathsf{const}(e) = v\} \\ & \cup (\mathit{if}\ (v = \top_v)\ \mathit{then}\ \{(\iota, x) \mid x \in \mathsf{Var}\}\ \mathit{else}\ \{\}) \end{aligned}$$

Finally, using $(\hookrightarrow_K^\dagger)$ and Θ_v, the algorithm constructs PA_K:

Definition 5.6 (PA_K). $\mathsf{PA}_K \in \mathbb{C} \to \mathbb{S}^\sharp$ is defined as follows:

$$\begin{aligned} \mathsf{PA}_K(c) = \ & \mathit{if}\ (c \notin C)\ \mathit{then}\ \bot \\ & \mathit{else}\ \lambda x. \bigsqcup \{v \in \mathbb{V} \mid \exists (c_0, x_0) \in \Theta_v . (c_0, x_0) \hookrightarrow_K^\dagger (c, x)\}. \end{aligned}$$

Then, PA_K is the solution of our pre-analysis:

LEMMA 5.7. *Let X be the least solution satisfying* (7). *Then,*

$$\mathsf{PA}_K(c) = \bigsqcup_{\kappa \in \mathbb{C}^*} X(c, \kappa).$$

PROOF. See A.2. □

5.2. Use of the Pre-Analysis Results

Using the pre-analysis results, we select queries that are likely to benefit from the increased context-sensitivity of the main analysis. Also, we collect calling contexts that are worth being distinguished during the main analysis. The collected contexts are used to construct a context selector K (Definition 5.15), which instructs how much context-sensitivity the main analysis should use for each procedure call. This main analysis with K is guaranteed to benefit from the increased context-sensitivity (Proposition 5.17).

Query selection. We first select queries that can benefit from increased context-sensitivity. Recall that a set of queries $\mathcal{Q} \subseteq \mathbb{C} \times \mathbb{S} \times \mathsf{Var}$ is given, that the analysis's job is to prove each $(c, s, x) \in \mathcal{Q}$ by computing an abstract state s' at the node c with $s' \sqsubseteq s$, and that the query is about the value of the variable x. In order to select queries whose resolution can benefit from increased context-sensitivity, we run the pre-analysis under full context-sensitivity K_∞. Let $\mathsf{PA}_{K_\infty} : \mathbb{C} \to \mathbb{S}^\sharp$ be the result of the pre-analysis. Using this result, we select queries as follows:

$$\begin{aligned}\mathcal{Q}^\sharp = \{(c,x) \in (\mathbb{C} \times \mathsf{Var}) \mid {} & \exists s \in \mathbb{S}. \\ & (c,s,x) \in \mathcal{Q} \;\wedge\; \forall s' \in \gamma(\mathsf{PA}_{K_\infty}(c)).\ s \sqcup s' \neq \top\}\end{aligned} \tag{9}$$

The first conjunct says that $(c, x) \in \mathcal{Q}^\sharp$ comes from some query $(c, s, x) \in \mathcal{Q}$, and the second conjunct expresses that according to the pre-analysis result, the main analysis does not lose too much information regarding this query. For instance, consider the case of interval analysis. In this case, we are usually interested in checking an assertion like $1 \leq x$ at c, which corresponds to a query (c, s, x) with the abstract state

$$s = (\lambda z.\ \text{if } (x = z) \text{ then } [1, \infty] \text{ else } \top).$$

Then, the second conjunct in (9) becomes equivalent to

$$\mathsf{PA}_{K_\infty}(c)(x) \sqsubseteq \bigstar.$$

That is, we select the query only when the pre-analysis estimates that the variable x will have at least a non-negative interval in the main analysis.

In the rest of this section, we assume for brevity that there is only one selected query $(c_q, x_q) \in \mathcal{Q}^\sharp$ in the program.

Building a context selector. Next, we construct a context selector $\mathsf{K} : \mathbb{F} \to \wp(\mathbb{C}_c^*)$. K is to answer which calling contexts the main analysis should distinguish in order to achieve most of the benefits of context sensitivity on the given query (c_q, x_q). Our construction considers the following proxy of this goal: which contexts should *the pre-analysis* distinguish to achieve the same precision on the selected query (c_q, x_q) as in the case of the full context-sensitivity? In this subsection, we will define a context selector K (Definition 5.15) that answers this question (Proposition 5.17).

We construct K in two steps. Before giving our construction, we remind the reader that the impact pre-analysis works on the value-flow graph $(\Theta, \hookrightarrow)$ of the program and computes the reachability relation $(\hookrightarrow^\dagger_{K_\infty}) \subseteq \Theta \times \Theta$ over the interprocedurally-valid paths.

The first step is to build a program slice that includes all the dependencies of the query (c_q, x_q). A query (c_q, x_q) depends on a vertex (c, x) in the value-flow graph if there exists an interprocedurally-valid path between (c, x) and (c_q, x_q) on the graph (i.e., $(c, x) \hookrightarrow^\dagger_{K_\infty} (c_q, x_q)$). Tracing the dependency backwards from the query eventually hits vertices with no predecessors. We call such vertices *sources* and denote their set by Φ.

Definition 5.8 (Φ). Sources Φ are vertices in Θ where dependencies begin:

$$\Phi = \{(c_0, x_0) \in \Theta \mid \neg(\exists (c, x) \in \Theta.\ (c, x) \hookrightarrow (c_0, x_0))\}.$$

The absence of predecessors implies that the abstract semantics at $(c_0, x_0) \in \Theta$ assigns a fixed constant abstract value to x_0 without using or joining other abstract values from vertices in Θ.Among vertices in Φ, we compute the set $\Phi_{(c_q,x_q)}$ of sources on which the query (c_q, x_q) depends.

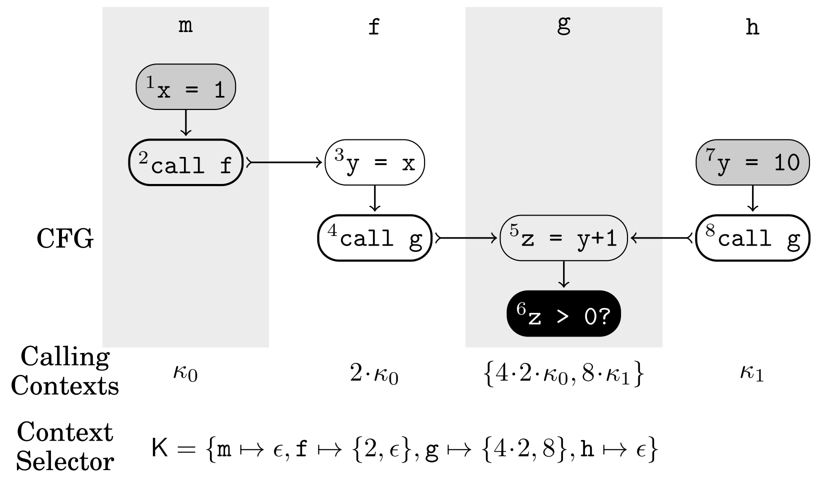

m
f
g
h
^{1}x = 1
^{2}call f
^{3}y = x
^{4}call g
^{5}z = y+1
^{6}z > 0?
^{7}y = 10
^{8}call g
CFG
Calling Contexts
κ_0
$2\cdot\kappa_0$
$\{4\cdot 2\cdot\kappa_0, 8\cdot\kappa_1\}$
κ_1
Context Selector
$\mathsf{K} = \{\mathtt{m} \mapsto \epsilon, \mathtt{f} \mapsto \{2, \epsilon\}, \mathtt{g} \mapsto \{4\cdot 2, 8\}, \mathtt{h} \mapsto \epsilon\}$

Fig. 2: Example context selector. Gray and black nodes in CFG are source and query points, respectively. The superscript in front of each command denotes the control point.

Definition 5.9 ($\Phi_{(c_q,x_q)}$). Sources on which the query (c_q, x_q) depends:

$$\Phi_{(c_q,x_q)} = \{(c_0, x_0) \in \Phi \mid (c_0, x_0) \hookrightarrow^{\dagger}_{K_\infty} (c_q, x_q)\}.$$

Example 5.10. Consider the control flow graph in Figure 2. Node 6 denotes the query point, i.e., $(c_q, x_q) = (6, \boldsymbol{z})$. The gray nodes represent the sources on which the query depends, i.e., $\Phi_{(6,\boldsymbol{z})} = \{(1, \boldsymbol{x}), (7, \boldsymbol{y})\}$.

For a source $(c_0, x_0) \in \Phi_{(c_q,x_q)}$ and an initial context κ_0 such that $(\iota, \epsilon) \to^*_{K_\infty} (c_0, \kappa_0)$, the following interprocedurally-valid path

$$((c_0, \kappa_0), x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q, \kappa_q), x_q) \tag{10}$$

represents a dependency path for the query (c_q, x_q). We denote the set of all dependency paths for the query by $\mathsf{Paths}_{(c_q,x_q)}$.

Definition 5.11 ($\mathsf{Paths}_{(c_q,x_q)}$). The set of all dependency paths for the query (c_q, x_q) is defined as follows:

$$\begin{aligned}\mathsf{Paths}_{(c_q,x_q)} = \{&((c_0, \kappa_0), x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q, \kappa_q), x_q) \\ &\mid (c_0, x_0) \in \Phi_{(c_q,x_q)} \ \wedge\ (\iota, \epsilon) \to^*_{K_\infty} (c_0, \kappa_0)\}.\end{aligned}$$

$\mathsf{Paths}_{(c_q,x_q)}$ is the program slice we intend to construct in this step.

Example 5.12. In Figure 2, suppose that κ_0 and κ_1 are the initial contexts of procedures *m* and *h*, respectively. For source $(1, \boldsymbol{x})$, we find the following dependency path to the query $(6, \boldsymbol{z})$:

$$\begin{aligned}p_1 = ((1, \kappa_0), \boldsymbol{x}) &\hookrightarrow_{K_\infty} ((2, \kappa_0), \boldsymbol{x}) \hookrightarrow_{K_\infty} ((3, 2\cdot\kappa_0), \boldsymbol{y}) \\ &\hookrightarrow_{K_\infty} ((4, 2\cdot\kappa_0), \boldsymbol{y}) \hookrightarrow_{K_\infty} ((5, 4\cdot 2\cdot\kappa_0), \boldsymbol{z}) \hookrightarrow_{K_\infty} ((6, 4\cdot 2\cdot\kappa_0), \boldsymbol{z})\end{aligned}$$

and, for source $(7, \boldsymbol{y})$, we find the following path to $(6, \boldsymbol{z})$:

$$\begin{aligned}p_2 = ((7, \kappa_1), \boldsymbol{y}) &\hookrightarrow_{K_\infty} ((8, \kappa_1), \boldsymbol{y}) \hookrightarrow_{K_\infty} ((5, 8\cdot\kappa_1), \boldsymbol{z}) \\ &\hookrightarrow_{K_\infty} ((6, 8\cdot\kappa_1), \boldsymbol{z}).\end{aligned}$$

Then, $\mathsf{Paths}_{(6,\boldsymbol{z})} = \{p_1, p_2\}$.

The next step is to compute calling contexts that should be treated precisely. Consider a dependency path from $\mathsf{Paths}_{(c_q,x_q)}$:

$$((c_0,\kappa_0),x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q,\kappa_q),x_q) \tag{11}$$

where $\kappa_0,\kappa_1,\ldots,\kappa_q$ are the calling contexts appeared in the (fully context-sensitive) pre-analysis. Instead, we are interested in partial contexts that represent the "difference" between κ_i and κ_0. Intuitively, if κ_0 is a suffix of κ_i, i.e., $\kappa_i = \kappa_i' \cdot \kappa_0$, the partial context for κ_i is defined as κ_i'. Formally, we define the partial calling contexts of κ_i as

$$\kappa_i \ominus \kappa_0 = \kappa_i - \mathsf{suffix}(\kappa_i,\kappa_0)$$

where $\mathsf{suffix}(\kappa_1,\kappa_2)$ is the longest common suffix of κ_1 and κ_2. The definition of suffix is as follows:

$$\mathsf{suffix}(\kappa_1,\kappa_2) = \begin{cases} \mathsf{suffix}(\kappa_1',\kappa_2')\cdot c & \text{when } \kappa_1=\kappa_1'\cdot c \ \wedge\ \kappa_2=\kappa_2'\cdot c \\ \epsilon & \text{otherwise} \end{cases}$$

For example, when κ_i is a suffix of κ_0, we use ϵ as the partial context for κ_i: if $\kappa_0 = c_2\cdot c_1$ and $\kappa_i = c_1$, then $\kappa_i \ominus \kappa_0 = \epsilon$. Suppose that κ_i and κ_0 are not a suffix of each other, for instance $\kappa_0 = c_2\cdot c_1$ and $\kappa_i = c_3\cdot c_1$. In this case, $\kappa_i\ominus\kappa_0 = c_3$.

In summary, for the path in (11), collecting contexts

$$\{\kappa_0\ominus\kappa_0,\ldots,\kappa_q\ominus\kappa_0\}$$

give all the necessary partial calling contexts, where each $\kappa_i\ominus\kappa_0$ belongs to the calling contexts of procedure $\mathsf{fid}(c_i)$. Thus, we define the context selector for the dependency path (11) as follows:

Definition 5.13 (K_p*, Context Selector for Path* p). Let p be a dependency path from a source (c_0,x_0) to query (c_q,x_q):

$$p = ((c_0,\kappa_0),x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q,\kappa_q),x_q),$$

where κ_0 is an initial context at c_0 such that $(\iota,\epsilon) \rightarrow^*_{K_\infty} (c_0,x_0)$. The context selector K_p for the path is defined as,

$$\mathsf{K}_p = \lambda f.\ \{\kappa_i\ominus\kappa_0 \mid \mathsf{fid}(c_i)=f\ \wedge\ ((c_i,\kappa_i),\_)\in p\}.$$

Example 5.14. From the path p_1 in Example 5.12, the collection of κ_i is $\{\kappa_0, 2\cdot\kappa_0, 4\cdot 2\cdot\kappa_0\}$ (see Figure 2). Hence, the collection of $\kappa_i\ominus\kappa_0$ is $\{\epsilon, 2, 4\cdot 2\}$, where ϵ belongs to procedure *m*, 2 to *f*, and $4\cdot 2$ to *g*. Similar for path p_2. Thus, K_{p_1} and K_{p_2} are:

$$\mathsf{K}_{p_1} = \begin{bmatrix} m \mapsto \{\epsilon\} \\ f \mapsto \{2\} \\ g \mapsto \{4\cdot 2\} \end{bmatrix} \quad \mathsf{K}_{p_2} = \begin{bmatrix} h \mapsto \{\epsilon\} \\ g \mapsto \{8\} \end{bmatrix}$$

Then, the final context selector K is the union of K_p's:

Definition 5.15 (K*, Context Selector*). Let (c_q,x_q) be a query. The context selector $\mathsf{K}\in\mathbb{F}\rightarrow\wp(\mathbb{C}_c^*)$ for our selective analysis is:

$$\mathsf{K}(f) \ = \ \mathcal{E}(f)\cup\bigcup\{\mathsf{K}_p(f)\mid p\in\mathsf{Paths}_{(c_q,x_q)}\} \tag{12}$$

where $\mathcal{E}(f)=\{\epsilon\}$ if $f\neq\mathsf{fid}(c_q)$; and otherwise, $\mathcal{E}(f)=\emptyset$.

Note that $\mathsf{K}(f)$ does not include ϵ when f is the procedure enclosing the query. This is because we have completely considered the calling contexts that are relevant to the query, and hence there are no other contexts to merge into ϵ. Other procedures, however, may require ϵ in the selective context-sensitive analysis.

Example 5.16. The final context selector for the program in Figure 2:

$$\mathsf{K} = \begin{bmatrix} m & \mapsto & \{\epsilon\} \\ f & \mapsto & \{2, \epsilon\} \\ g & \mapsto & \{4 \cdot 2, 8\} \\ h & \mapsto & \{\epsilon\} \end{bmatrix}$$

In practice, building the context selector K can be efficiently implemented. Given a query (c_q, x_q), we first identify its source nodes $(c_0, x_0) \in \Phi_{(c_q, x_q)}$ in the value-flow graph. Then, we explore the all interprocedurally-valid paths from sources (c_0, x_0) to (c_q, x_q) while collecting call sequences along the paths. The collected call sequences are the partial contexts $\{\kappa_0 \ominus \kappa_0, \ldots, \kappa_q \ominus \kappa_0\}$. Then, we build the context selector from this set. Given that there are no recursive procedures, these steps are computed in finite time (we can ignore loops inside procedures, as calling contexts are updated at only procedure boundaries).

Running selective context-sensitive main analysis. Finally, we run the main analysis with selective context-sensitivity K defined by the result of the impact pre-analysis. The following proposition states that the pre-analysis-guided context-sensitivity (K) manages to pay off at the selective main analysis, although the pre-analysis is fully context-sensitive and the main analysis is not.

PROPOSITION 5.17 (IMPACT REALIZATION). *Let* $\mathsf{PA}_{K_\infty} \in \mathbb{C} \to \mathbb{S}^\sharp$ *be the result of the impact pre-analysis (Definition 5.6). Let* $q \in \mathcal{Q}^\sharp$ *be a selected query* (9). *Let* K *be the context selector for* q *(Definition 5.15) defined using the pre-analysis result* PA_{K_∞}. *Let* $\mathsf{MA}_\mathsf{K} \in \mathbb{C}_K \to \mathbb{S}$ *be the main analysis result with the context selector* K. *Then, the selective main analysis is at least as precise as the fully context-sensitive pre-analysis for the selected query* q*:*

$$\mathsf{MA}_\mathsf{K} \sqsubseteq_q \mathsf{PA}_{K_\infty}$$

where $\mathsf{MA}_\mathsf{K} \sqsubseteq_q \mathsf{PA}_{K_\infty}$ *iff* $(q \overset{let}{=} (c, x))$

$$\forall \kappa \in \mathsf{K}(\mathsf{fid}(c)).\ \mathsf{MA}_\mathsf{K}(\kappa, c) \in \gamma(\top[x \mapsto \mathsf{PA}_{K_\infty}(c)(x)]).$$

PROOF. See A.1. □

This impact realization holds thanks to two key properties. First, our selective context-sensitivity K (Definition 5.15) distinguishes all the calling contexts that matter for the queries selected by the pre-analysis. Second, the main analysis designed in Section 4 isolates these distinguished contexts from other undistinguished contexts (ϵ), ensuring that spurious flows caused by merging contexts never adversely affect the precision of the selected query.

6. APPLICATION TO OTHER SELECTIVE ANALYSES

A general principle behind our method is that we can selectively improve the precision of the analysis by using an impact pre-analysis that estimates the main static analysis of the maximal precision. In this section, we show how we could use the general principle to develop other selective program analyses. In Section 6.1, we design a a selective relational analysis with the octagon domain [Miné 2006], and in Section 6.2, we design a selective flow-sensitive analysis.

6.1. Selective Relational Analysis

Overview. A selective relational analysis aims to reduce the cost of the relational analysis by applying the relational analysis only when doing so is likely to benefit the final analysis precision. For instance, consider the following code:

```
1  int a = b;
2  int c = input();            // User input
3  for (i = 0; i < b; i++) {
4    assert (i < a);           // Query 1
5    assert (i < c);           // Query 2
6  }
```

Suppose that, at the beginning, the value of `b` is unknown. There are two queries in the program. At line 4, the query asks whether the value of `i` is less than the value of `a`. The other, at line 5, asks whether the value of `i` is less than the value of `c`. Note that the first query always holds, as `a` and `b` are equivalent (by line 1) and `i` is less than `b` according to the loop condition. On the other hand, the second query is not necessarily true because the value of `c` comes from the environment and might be greater than the value of `i`.

In a non-relational analysis, the first query would not be proved. For instance, suppose that we analyze the example program using the interval domain [Cousot and Cousot 1977]. The interval analysis is non-relational and cannot infer the equivalence of `a` and `b`. Instead, both `a` and `b` have $[-\infty, +\infty]$ in the analysis and `i` has $[0, +\infty]$ inside the loop. These analysis results are not strong enough to prove the query.

A fully relational octagon analysis, which tracks contraints of the form $\pm x \pm y \leq c$ (where $c \in \mathbb{Z} \cup \{\infty\}$) between *all* variables x and y, can prove the first query. The analysis infers constraints $\texttt{b} - \texttt{a} \leq 0$ at line 1 and $\texttt{i} - \texttt{b} \leq -1$ at line 3. Then, combining the two via a closure operation [Miné 2006], the analysis concludes that constraint $\texttt{i} - \texttt{a} \leq -1$ holds at line 4. More specifically, the fully relational octagon analysis computes the following table (i.e., difference bound matrix [Miné 2006]):

$$\begin{array}{c|c|c|c|c}
 & \texttt{a} & \texttt{b} & \texttt{c} & \texttt{i} \\
\hline
\texttt{a} & 0 & 0 & \infty & -1 \\
\hline
\texttt{b} & 0 & 0 & \infty & -1 \\
\hline
\texttt{c} & \infty & \infty & 0 & \infty \\
\hline
\texttt{i} & \infty & \infty & \infty & 0
\end{array} \tag{13}$$

where the bound c in constraint $x - y \leq c$ is stored at row y and column x in the table.[4] Note that the (`a`,`i`) entry of the table stores -1, which means that the analysis proves $\texttt{i} - \texttt{a} \leq -1$ at line 4.

However, this fully relational analysis tracks unnecessary relationships between variables, which are either irrelevant to the query or not beneficial to the analysis precision. For instance, it is sufficient to keep only the contraints between `a`, `b`, and `i` to prove the first query, but the analysis unnecessarily maintains other relationships such as one between `a` and `c`. Besides, tracking a relationship between, for example, `i` and `c` does not change the end result of the analysis because the second query is impossible to prove.

Our selective octagon analysis tracks octagon constraints only when doing so is likely to improve the precision that matters for resolving given queries. To achieve this goal, we use an impact pre-analysis that aims at estimating the behavior of the octagon analysis under its fully relational setting. More specifically, like the fully relational octagon analysis, the pre-analysis tracks constraints of the form $\pm x \pm y \leq a$ for *all* variables x and y but approximately tracks the bound; we use one of two abstract values $\bigstar$ and $\top$ as bound a, rather than all integers and ∞. Here $x + y \leq \top$ represents all octagon constraints of the form $x + y \leq c$ including the case that $c = \infty$, whereas

[4]For simplicity, we consider only constraints of the form $x - y \leq c$. In fact, the octagon analysis tracks constraints of both forms $x - y \leq c$ and $x + y \leq c$ and maintains a matrix of size $(2 \times |\mathsf{Var}|)^2$.

$x + y \sqsubseteq \bigstar$ means octagon constraints $x + y \leq c$ with integer constant c. This simple abstract domain is chosen because constant bound, not ∞, proves buffer-overrun properties. For instance, in our example program, the pre-analysis result at line 4 is given as follows:

	a	b	c	i
a	★	★	⊤	★
b	★	★	⊤	★
c	⊤	⊤	★	⊤
i	⊤	⊤	⊤	★

The table has the same size as the one in (13), but its entries represent the approximated bounds: abstract value $\bigstar$ denotes all integer constants, and $\top$ includes ∞ in addition to all the integer constants.

Next, using the pre-analysis results, we select variables whose relationships help improve the precision regarding given queries. We first identify queries (in our example, the first query) whose values are evaluated to $\bigstar$ using the pre-analysis results. Then, for each of selected queries, we do a dependency analysis to find out the variables whose relationships should be tracked together for the main analysis to answer query. For instance, consider that the constraint regarding the first query is $\texttt{i} - \texttt{a} \sqsubseteq \bigstar$. Our dependency analysis figures out that the constraint was derived in the pre-analysis by combining two constraints $\texttt{i} - \texttt{b} \sqsubseteq \bigstar$ and $\texttt{b} - \texttt{a} \sqsubseteq \bigstar$ in its closure operation. Therefore, the dependency analysis concludes that the main analysis should be able to derive three relationships $\texttt{i} - \texttt{a} \sqsubseteq \bigstar$, $\texttt{i} - \texttt{b} \sqsubseteq \bigstar$, and $\texttt{b} - \texttt{a} \sqsubseteq \bigstar$ to prove the first query. Based on this conclusion, our selective octagon analysis decides to track the relationships between variables a, b, and i.

In the rest of this section, we formalize the key aspects of our selective octagon analysis.

Selective octagon analysis via variable packing. We first specify selective octagon analyses for the following simple commands:

$$cmd \ \rightarrow \ x := y + k \mid x :=?$$

where $k \in \mathbb{Z}$ is a positive integer and $?$ models arbitrary integers. We use Miné's definitions [Miné 2006] of the octagon domain $\mathbb{O}$ and abstract semantics $[\![cmd]\!] : \mathbb{O} \rightarrow \mathbb{O}$ of primitive commands; we consider the positive form x and negative form $\bar{x}$ for each variable x and represent an octagon domain element $o \in \mathbb{O}$ by a $2|\mathsf{Var}| \times 2|\mathsf{Var}|$ matrix where each entry $o_{xy} \in \mathbb{Z} \cup \{+\infty\}$ stores the upper bound of $y - x$. The definition of $[\![cmd]\!]$ for our commands can be found at [Miné 2006].

With $\mathbb{O}$ and $[\![cmd]\!]$, we define the domain of *packed octagons* that assign an octagon to a subset of variables, which we call *pack*. An octagon of a pack expresses only the constraints of the variables in that pack. We call $\Pi \subseteq \wp(\mathsf{Var})$ of sets of variables *packing configuration*, such that $\bigcup \Pi = \mathsf{Var}$. The packed octagon domain $\mathbb{PO}(\Pi)$ parameterized by packing configuration Π is then defined as $\mathbb{PO}(\Pi) = \Pi \rightarrow \mathbb{O}$. We extend the abstract semantics $[\![cmd]\!] : \mathbb{O} \rightarrow \mathbb{O}$ of command cmd to $[\![cmd]\!]^{\Pi} : \mathbb{PO}(\Pi) \rightarrow \mathbb{PO}(\Pi)$ as follows:

$$[\![c]\!]^{\Pi}(po) = \lambda\pi \in \Pi.$$
$$\begin{cases} [\![x := y + k]\!](po(\pi)) & (c = x := y + k \ \wedge \ x \in \pi \ \wedge \ y \in \pi) \\ [\![x :=?]\!](po(\pi)) & (c = x := y + k \ \wedge \ x \in \pi \ \wedge \ y \notin \pi) \\ [\![x :=?]\!](po(\pi)) & (c = x :=? \ \wedge \ x \in \pi) \\ po(\pi) & \textit{otherwise} \end{cases}$$

The extended abstract semantics is essentially the same except it forgets all the relationships of the assignee x (the second case) when the pack is missing one variable involved in the octagonal constraint. The abstract semantics of program in $\mathbb{D} = \mathbb{C} \rightarrow \mathbb{PO}(\Pi)$ is defined as the least fixpoint of abstract transfer function $F^{\Pi} : \mathbb{D} \rightarrow \mathbb{D}$, which is defined as usual.

The selectivity of the analysis is governed by the configuration Π. For instance, with $\Pi = \{\{x\} \mid x \in \mathsf{Var}\}$, the analysis degenerates to a non-relational analysis. With $\Pi = \{\mathsf{Var}\}$, the analysis becomes a fully relational analysis. Our goal is to find a cost-effective Π by using an impact pre-analysis.

Impact pre-analysis. Second, we formally define the impact pre-analysis. The meaning of our abstract values ($\mathbb{V} = \{\bigstar, \top\}$) is described by $\gamma_{\mathbb{V}} : \mathbb{V} \rightarrow \wp(\mathbb{Z} \cup \{+\infty\})$ such that

$$\begin{aligned}\gamma_{\mathbb{V}}(\bigstar) &= \mathbb{Z}\\ \gamma_{\mathbb{V}}(\top) &= \mathbb{Z} \cup \{+\infty\}\end{aligned}$$

The abstract state $\mathbb{O}^\sharp = \{\bot^\sharp\} \cup \mathbb{V}^{2|\mathsf{Var}| \times 2|\mathsf{Var}|}$ of our pre-analysis is the set of matrices whose entries are in $\mathbb{V}$. An abstract state $o^\sharp \in \mathbb{O}^\sharp$ denotes a set of octagons: we define $\gamma : \mathbb{O}^\sharp \rightarrow \wp(\mathbb{O})$ as follows:

$$\gamma(o^\sharp) = \{o \in \mathbb{O} \mid \forall i, j.\ o_{ij} \in \gamma_{\mathbb{V}}(o^\sharp_{ij})\}$$

The abstract semantics $[\![cmd]\!]^\sharp : \mathbb{O}^\sharp \rightarrow \mathbb{O}^\sharp$ of each primitive command cmd of the pre-analysis is defined as an over-approximation of the abstract semantics of the main analyses: e.g.,

$$\big([\![x := ?]\!]^\sharp(o^\sharp)\big)_{ij} = \begin{cases} \bigstar & (i = j = x \text{ or } i = j = \bar{x}) \\ \top & (i \notin \{x, \bar{x}\} \text{ or } j \notin \{x, \bar{x}\}) \\ o^\sharp_{ij} & \textit{otherwise} \end{cases}$$

The abstract domain of the pre-analysis is $\mathbb{D}^\sharp = \mathbb{C} \rightarrow \mathbb{O}^\sharp$ and the pre-analysis result is defined as the least fixpoint of semantic function $F^\sharp : \mathbb{D}^\sharp \rightarrow \mathbb{D}^\sharp$, which is defined as usual.

Use of pre-analysis results. From the pre-analysis results ($\mathsf{lfp}F^\sharp$), we construct Π as follows. Assume that a set $\mathcal{Q} \subseteq \mathbb{C} \times \mathsf{Var} \times \mathsf{Var}$ of relational queries is given in the program. A query $(c, x, y) \in \mathcal{Q}$ represents a predicate $y - x < 0$ at program point c and we say that $o \in \mathbb{O}$ proves the query when $o_{xy} \leq -1$. We first select a set $\mathcal{Q}^\sharp$ of queries that are judged promising by the pre-analysis:

$$\mathcal{Q}^\sharp = \{(c, x, y) \in \mathcal{Q} \mid (\mathsf{lfp}F^\sharp)(c) \neq \bot^\sharp \wedge (\mathsf{lfp}F^\sharp)(c)_{xy} = \bigstar\}.$$

Next, for each selected query $(c, x, y) \in \mathcal{Q}^\sharp$, we compute the pack $\pi_{(c,x,y)} \subseteq \mathsf{Var}$ of necessary variables using dependency analysis, which is simultaneously done with the pre-analysis as follows: let $\mathbb{V}^\natural$ be $\mathbb{V} \times \wp(\mathsf{Var})$ and $\mathbb{O}^\natural$ be the set of $2|\mathsf{Var}| \times 2|\mathsf{Var}|$ matrices over $\mathbb{V}^\natural$. The idea is to over-approximate the involved variables for each octagon constraint in the second component of $\mathbb{V}^\natural$. The abstract semantics $[\![\cdot]\!]^\natural : \mathbb{O}^\natural \rightarrow \mathbb{O}^\natural$ is the same as $[\![\cdot]\!]^\sharp$ except that it also maintains the involved variables: e.g.,

$$\big([\![x := ?]\!]^\natural(o^\natural)\big)_{ij} = \begin{cases} \big(\bigstar, \{i, j\}\big) & (i = j = x \text{ or } i = j = \bar{x}) \\ (\top, \mathsf{Var}) & (i \notin \{x, \bar{x}\} \text{ or } j \notin \{x, \bar{x}\}) \\ {o^\natural}_{ij} & \textit{otherwise} \end{cases}$$

Let $F^\natural : (\mathbb{C} \rightarrow \mathbb{O}^\natural) \rightarrow (\mathbb{C} \rightarrow \mathbb{O}^\natural)$ be the abstract transfer function and $\mathsf{lfp}F^\natural$ be its least fixpoint. Then, the pack $\pi_{(c,x,y)}$ is defined as S such that $\big((\mathsf{lfp}F^\natural)(c)\big)_{xy} = \big(\bigstar, S\big)$. Finally, we extract the packing configuration Π using $\pi_{(c,x,y)}$ as follows:

$$\Pi = \{\pi_{(c,x,y)}\} \cup \big\{\{z\} \mid z \in \mathsf{Var} \setminus \pi_{(c,x,y)}\big\}. \tag{14}$$

Selective main octagon analysis. We run the selective octagon analysis with the packing configuration in (14).

PROPOSITION 6.1 (IMPACT REALIZATION FOR SELECTIVE RELATIONAL ANALYSIS). *Let $\pi_{(c,x,y)}$ be the pack for query (c, x, y) defined by the result of our impact pre-analysis. Let Π be the packing configuration for $\pi_{(c,x,y)}$, which is defined in* (14)*. Let F^{Π} be the transfer function of the selective octagon analysis with the Π. Then,*

$$\left((\mathsf{lfp} F^{\Pi})(c)(\pi_{(c,x,y)})\right)_{xy} \neq +\infty.$$

6.2. Selective Flow-Sensitive Analysis

Overview. A selective flow-sensitive analysis aims to apply flow-sensitivity only when it is likely to benefit the final analysis results. This technique is useful when we need to reduce the analysis cost while maintaining the benefits of flow-sensitivity. For instance, consider the following code, adapted from `barcode-0.96`:

```
1  i = 0;
2  codes[i++] = 1;                      /* buffer-access 1 */
3  for (s = bc->ascii; ...; s++)
4    *s = ...;                          /* buffer-access 2 */
```

Suppose that a (fully) flow-sensitive analysis can prove the buffer-access safety at line 2 but the access at line 4 cannot be proved safe even with flow-sensitivity due to the analysis' inherent incompleteness (e.g., `bc->ascii` is unknown during the analysis). In this case, we aim to apply flow-sensitivity only to program points (lines) 1 and 2, and analyze others (lines 3 and 4) flow-insensitively.

Next, we formalize the idea of selective flow-sensitivity and describe how to design an impact pre-analysis for finding program points that need flow-sensitivity.

A selective flow-sensitive analysis with parameter π. We assume that a program is represented by a control flow graph $(\mathbb{C}, \rightarrow)$ where $\mathbb{C}$ denotes the finite set of nodes and $(\rightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ denotes the control flows between nodes. Let $\iota \in \mathbb{C}$ be the initial program point of the program. We associate a primitive command with each node $c \in \mathbb{C}$ of our control flow graph, and denote it by $\mathsf{cmd}(c)$. The set of possible primitive commands is specified by the following grammar:

$$cmd \ \rightarrow \ skip \mid x := e$$

where e is an arithmetic expression:

$$e \ \rightarrow \ n \mid x \mid e + e \mid e - e$$

We denote the set of all program variables by Var.

We define a selective flow-sensitive analysis as a special instance of the trace partitioning [Rival and Mauborgne 2007]. Let Δ be the set of partitioning indicies and

$$\pi : \Delta \rightarrow \wp(\mathbb{C})$$

be the partitioning function such that $\mathbb{C} = \bigcup_{i \in \Delta} \pi(i)$. Given partitioning index $i \in \Delta$, $\pi(i)$ denotes the set of program points that are partitioned by index i. The analysis keeps an abstract state for each partitioning index (rather than for each program point). Thus, the abstract domain is a map from partitioning indicies to abstract states, i.e.,

$$\mathbb{D} = \Delta \rightarrow \mathbb{S}$$

and the abstract transfer function $F : (\Delta \to \mathbb{S}) \to (\Delta \to \mathbb{S})$ is defined as follows:

$$F(X)(i) = \bigsqcup_{c \in \pi(i)} [\![\mathsf{cmd}(c)]\!](\bigsqcup_{c_0 \to c} \bigsqcup_{c_0 \in \pi(i_0)} X(i_0)) \tag{15}$$

Note that the analysis merges the effects of all commands denoted by index i, and π determines the level of flow-sensitivity. For instance, when $\Delta = \mathbb{C}$ and $\pi = \lambda c.\{c\}$, the analysis becomes the ordinary flow-sensitive analysis, and when Δ is a singleton set and π maps the single index to the entire set of program points, the analysis gets instantiated to the conventional flow-insensitive analysis. Our goal is to find a parameter π that yields an analysis located in a sweet spot between the two extremes.

Impact pre-analysis for finding π. We find such a flow-sensitivity parameter π by using an impact pre-analysis. This pre-analysis focuses only on estimating the impact of the flow-sensitivity. To do so, the pre-analysis runs with full flow-sensitivity but with a simpler abstract domain than that of the main analysis. Let $\mathbb{S}^\sharp$ be the abstract state and $([\![-]\!]^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp)$ be the abstract semantics of the pre-analysis, which is connected with the main analysis by the soundness condition in Section 5.1. In addition, in order to reuse the efficient reachability analysis algorithm (Section 5.1), we further require the analysis to satisfy the computability condition in Section 5.1, which is specified as follows:

(1) The abstract states are $\bot$ or functions from program variables to abstract values:
$$\mathbb{S}^\sharp = \{\bot\} \cup (\mathsf{Var} \to \mathbb{V})$$
where $\mathbb{V}$ is a finite complete lattice:
$$(\mathbb{V}, \sqsubseteq_v, \bot_v, \top_v, \sqcup_v, \sqcap_v).$$
For instance, in experiments, we use the $\mathbb{V} = \{\top, \bigstar, \bot\}$ domain.

(2) An abstract semantics of primitive commands is defined as follows:
$$\begin{aligned} [\![\mathit{skip}]\!]^\sharp(s) &= s \\ [\![x := e]\!]^\sharp(s) &= \begin{cases} s[x \mapsto [\![e]\!]^\sharp(s)] & (s \neq \bot) \\ \bot & (s = \bot) \end{cases} \end{aligned}$$
where $[\![e]\!]^\sharp$ has the following form: for every $s \neq \bot$,
$$[\![e]\!]^\sharp(s) = s(x_1) \sqcup \ldots \sqcup s(x_n) \sqcup v$$
for some variables $x_1, \ldots, x_n$ and an abstract value $v \in \mathbb{V}$, all of which are fixed for the given e. We denote these variables and the value by
$$\mathsf{var}(e) = \{x_1, \ldots, x_n\}, \qquad \mathsf{const}(e) = v.$$

Since the pre-analysis is fully flow-sensitive, the partitioning indicies are program points: we use the following partitioning indices and function:

$$\Delta = \mathbb{C}, \qquad \pi = \lambda c.\{c\}$$

Thus, the abstract domain is a map from program points to states:

$$\mathbb{D}^\sharp = \mathbb{C} \to \mathbb{S}^\sharp$$

and the abstract semantic function $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ is defined as follows:

$$F^\sharp(X)(c) = [\![\mathsf{cmd}(c)]\!]^\sharp(\bigsqcup_{c_0 \to c} X(c_0)). \tag{16}$$

We compute $\mathsf{lfp} F^\sharp$ via a simplified version of our reachability algorithm. Unlike the reachability algorithm in Section 5.1, in this case, we do not need to compute the interprocedurally-valid reachability but only a plain reachability over the value-flow graph (with the assumption that the analysis is context-insensitive). As in the pre-analysis algorithm of the selective context-sensitivity, we construct value-flow graph $(\Theta, \hookrightarrow)$:

$$\Theta = \mathbb{C} \times \mathsf{Var}, \qquad (\hookrightarrow) \subseteq \Theta \times \Theta$$

where $(c, x) \hookrightarrow (c', x')$ iff

$$\begin{cases} c \to c' \ \wedge \ x = x' & (\mathsf{cmd}(c') = \mathit{skip}) \\ c \to c' \ \wedge \ x = x' & (\mathsf{cmd}(c') = y := e \ \wedge \ y \neq x') \\ c \to c' \ \wedge \ x \in \mathsf{var}(e) & (\mathsf{cmd}(c') = y := e \ \wedge \ y = x') \end{cases}$$

Then we find a set Θ_v of generators for each abstract value v in $\mathbb{V}$. Generators for v are vertices in Θ whose commands join v in their abstract semantics:

$$\begin{aligned} \Theta_v = & \{(c, x) \mid \mathsf{cmd}(c) = x := e \wedge \mathsf{const}(e) = v\} \\ & \cup (\textit{if } (v = \top_v) \textit{ then } \{(\iota, x) \mid x \in \mathsf{Var}\} \textit{ else } \{\}) \end{aligned}$$

Finally, we compute the pre-analysis results $\mathsf{PA} \in \mathbb{C} \to \mathbb{S}^\sharp$ as follows:

$$\mathsf{PA}(c) \ = \ \lambda x. \bigsqcup\{v \in \mathbb{V} \mid \exists (c_0, x_0) \in \Theta_v. \ (c_0, x_0) \hookrightarrow^* (c, x)\}.$$

The following lemma shows that PA is indeed a solution of the pre-analysis.

LEMMA 6.2. $\forall c \in \mathbb{C}. \ \mathsf{PA}(c) = (\mathsf{lfp} F^\sharp)(c)$.

Use of the pre-analysis results. Now we use the pre-analysis results $\mathsf{PA}(c)$ to construct the flow-sensitivity parameter π. We first select queries that can benefit from increased flow-sensitivity. As in the selective context-sensitivity method, we select $\mathcal{Q}^\sharp \subseteq \mathbb{C} \times \mathsf{Var}$ from given queries $\mathcal{Q} \subseteq \mathbb{C} \times \mathbb{S} \times \mathsf{Var}$:

$$\begin{aligned} \mathcal{Q}^\sharp = \{(c, x) \in (\mathbb{C} \times \mathsf{Var}) \mid & \ \exists s \in \mathbb{S}. \\ & (c, s, x) \in \mathcal{Q} \wedge \ \forall s' \in \gamma(\mathsf{PA}(c)). \ s \sqcup s' \neq \top\} \end{aligned}$$

For simplicity, in the rest of this section, we assume that there is only one selected query $(c_q, x_q) \in \mathcal{Q}^\sharp$.

Next, we construct a partitioning function $\pi : \Delta \to \wp(\mathbb{C})$. Intuitively, π prescribes a set of control flows that need to be distinguished in order to maintain the precision of the full flow-sensitivity regarding the selected query (c_q, x_q).

The first step is to compute a program slice that includes all the dependencies of the selected query (c_q, x_q). In this case, the query (c_q, x_q) depends on a vertex (c, x) if (c_q, x_q) is reachable from (c, x) over the value-flow graph, i.e., $(c, x) \hookrightarrow^* (c_q, x_q)$. Let Φ be the set of vertices where dependencies begin:

$$\Phi = \{(c_0, x_0) \in \Theta \mid \nexists (c, x) \in \Theta. \ (c, x) \hookrightarrow (c_0, x_0)\}$$

and let $\mathcal{S}_{(c_q, x_q)}$ be the set of vertices on the paths from Φ to (c_q, x_q):

$$\mathcal{S}_{(c_q, x_q)} = \{c \in \mathbb{C} \mid (c_0, x_0) \hookrightarrow^* (c, x) \hookrightarrow^* (c_q, x_q) \ \wedge \ (c_0, x_0) \in \Phi\}$$

Then, we define $\Delta = \mathcal{S}_{(c_q, x_q)} \cup \{\bullet\}$ and the partitioning function π as follows:

$$\pi(i) = \begin{cases} \{i\} & i \in \mathcal{S}_{(c_q, x_q)} \\ \mathbb{C} \setminus \mathcal{S}_{(c_q, x_q)} & \text{otherwise (i.e., when } i = \bullet) \end{cases} \tag{17}$$

where $\bullet$ denotes all the other program points not included in $\mathcal{S}_{(c_q, x_q)}$. Intuitively, the partitioning function says that we flow-sensitively analyze the program points on which the query depends and apply flow-insensitivity to other program points.

Selective flow-sensitive main analysis. We run the main analysis (15) with the partitioning function π in (17). The following lemma shows that the impact estimation of the pre-analysis manages to pay off at the selective flow-sensitive main analysis.

PROPOSITION 6.3 (IMPACT REALIZATION FOR SELECTIVE FLOW-SENSITIVITY). *Let* MA_π *be the results of the main analysis in* (15) *with* π *in* (17). *Let* PA *be the result of the pre-analysis in* (16). *Let* (c_q, x_q) *be a query. Then,*

$$\mathsf{MA}_\pi \sqsubseteq_q \mathsf{PA}$$

where $\mathsf{MA}_\pi(c_q) \sqsubseteq_q \mathsf{PA}$ *iff (let* $q = (c_q, x_q)$*)*

$$\mathsf{MA}_\pi(c_q) \in \gamma(\top[x_q \mapsto \mathsf{PA}(c_q)(x_q)]).$$

7. EXPERIMENTS

We evaluate the effectiveness of our method by experiments with SPARROW, an interval domain–based static analyzer for C programs [Oh et al.]. We avoid direct comparisons with other approaches since, to the best of our knowledge, there has been no related work directly comparable with our approach. The most similar approach to ours is the one due to [Guyer and Lin 2003a], but the approach is particularly designed for a pointer analysis and provides no clue to generalize to other types of analyses we handle. Instead, we study the effectiveness by the difference in the precision and the overhead within the same static analyzer.

7.1. Setting

SPARROW is a buffer-overrun analyzer that supports full set of the C language. The baseline analyzer performs a flow-sensitive and context-insensitive analysis, and tracks both numeric and pointer values. For numeric values, it uses the interval domain by default (alternatively, it can use the octagon domain). In addition to the interval domain, the analysis uses an allocation-site–based heap abstraction for dynamic memory allocation. The analysis is field-sensitive. Thus, an abstract state of the analysis is a map from abstract locations (program variables, allocation-sites, and structure fields) to abstract values of intervals and points-to sets. These numeric and pointer values are analyzed simultaneously in a single fixpoint iteration. The analysis computes an abstract state for each program point. More details on the analysis can be found in our previous papers [Oh and Yi 2013; Oh et al. 2012].

We have run the analysis for various software packages from the GNU open-source projects. The analysis is global: the entire program is analyzed starting from the `main` procedure. For procedure calls whose bodies are not available in source code, we use handcrafted function stubs for standard library calls and otherwise we assume that the procedure calls return arbitrary values and have no side-effects. All experiments were done on a Linux 2.6 system running on a single core of Intel 3.07GHz box with 24GB of memory.

In experiments, we used the analysis results to prove the safety of buffer accesses in the program. Given a buffer access `x[i]` and interval analysis results $a \leq \texttt{i} \leq b$ and $c \leq \texttt{sizeof(x)} \leq d$, where a, b, c, d are constant integers, the buffer access is safe if $0 \leq a \ \wedge \ b < c$ holds; otherwise, the analysis raises a buffer-overrun alarm.

7.2. Selective Context-Sensitive Analysis

On top of the baseline analyzer, we have implemented the impact pre-analysis in Example 5.2 and extended the baseline analysis to be selectively context-sensitive. The pre-analysis gives a set of call sequences that should be treated context-sensitively. This information guides the main analysis to perform context-sensitive analysis in a selective manner. In Section 3, we assumed there are no recursive procedures in the

Program	**LOC**	**Proc**	**Context-Insensitive**	
			#alarm	**time**
spell-1.0	2,213	31	58	0.6
bc-1.06	13,093	134	606	14.0
tar-1.17	20,258	222	940	42.1
less-382	23,822	382	654	123.0
sed-4.0.8	26,807	294	1,325	107.5
make-3.76	27,304	191	1,500	84.4
grep-2.5	31,495	153	735	12.1
wget-1.9	35,018	434	1,307	69.0
a2ps-4.14	64,590	980	3,682	118.1
bison-2.5	101,807	1,427	1,894	136.3
Total	346,407	4,248	12,701	707.1

Program	**Our Selective Context-Sensitive Analysis**						**Alarm**	**Overhead**	
	#alarm	**pre**	**main**	**total**	**#selected call-sites**	$\rightsquigarrow$	**reduction**	**pre**	**main**
spell-1.0	30	0.1	0.8	0.9	25 / 124 (20.2 %)	3	48.3%	16.7%	33.3%
bc-1.06	483	1.9	14.3	16.2	29 / 777 (3.7 %)	2	20.3%	13.6%	2.1%
tar-1.17	799	5.4	41.8	47.2	51 / 1213 (4.2 %)	3	15.0%	12.8%	−0.7%
less-382	562	3.3	163.1	166.4	51 / 1,522 (3.4 %)	4	14.1%	2.7%	32.6%
sed-4.0.8	1,238	7.4	110.2	117.6	25 / 868 (2.9 %)	3	6.6%	6.9%	2.5%
make-3.76	1,028	7.1	99.1	106.2	67 / 1,050 (6.4 %)	3	31.5%	8.4%	17.4%
grep-2.5	653	2.4	13.5	15.9	33 / 530 (6.2 %)	3	11.2%	19.8%	11.6%
wget-1.9	942	12.5	69.6	82.1	79 / 1,973 (4.0 %)	5	27.9%	18.1%	0.9%
a2ps-4.14	2,121	29.5	148.2	177.7	237 / 2,450 (9.7 %)	9	42.4%	25.0%	25.5%
bison-2.5	1,742	34.6	138.8	173.4	173 / 2,038 (8.5 %)	4	8.0%	25.4%	1.8%
Total	9,598	104.2	799.4	903.6	770 / 12,545 (6.1 %)		**24.4%**	**14.7%**	**13.1%**

Table I: Performance comparison between context-insensitive analysis and our selective context-sensitive analysis. **LOC** reports lines of code before pre-processing. **Proc** shows the number of procedures in the programs. **#alarm** reports the number of buffer-overrun alarms raised by the analyses. **pre** reports the time spent for running the pre-analysis (including query selection and building context selector) and **main** reports the time spent by the main analysis of our approach. Each entry a/b (c%) in column **#selected call-sites** means that, among b call-sites in the program, a call-sites are selected for context-sensitivity by our pre-analysis and the selection ratio is c%. $\rightsquigarrow$ reports the maximum call-depth prescribed by the pre-analysis. **Overhead**: **pre** shows the pre-analysis overhead and **main** reports the cost increase in the main analysis due to increased context-sensitivity, compared to the context-insensitive analysis.

program. In fact, the pre-analysis and main analysis run in the presence of recursion. We assumed so because we do not increase context-sensitivity for queries involved in the recursive procedures. In Section 5.2, we considered only one query; in implementation, the pre-analysis computes a single context selector K that specifies calling contexts for multiple queries. When analyzing a procedure under different calling contexts, we distinguish allocation sites for each context; that is, an allocation-site produces different abstract locations under different calling contexts.

Table I presents the performance of our selective context-sensitive analysis and compares it with the context-insensitive analysis. We measured the analysis precision by the number of buffer accesses (**#alarm**) that cannot be proven safe by the analysis. In the programs in Table I, there are a total of 83,776 buffer access expressions in the 10 programs.

Overall, our technique strikes a good balance between analysis precision and cost. In total, the context-insensitive interval analysis points out 12,701 buffer accesses as potential buffer-overrun errors. Our technique reduces the number down to 9,598 (24.4% reduction). In doing so, our technique increases the total analysis time from 707.1s to 903.6s (27.8% increase).

We observed that passing numeric values through long call chains is necessary in the interval analysis of C programs. For instance, in `a2ps-4.14`, among 1682 call sequences prescribed by our pre-analysis, 488 call sequences (29%) have length ≥ 3 and 208 call-sequences have length ≥ 5. Our observation does not contradict the folklore that tracking the call depth up to 2 (i.e., 2-callstrings approach) is sufficient for most applications. This folklore is mostly based on experiences with the data-flow analyses for object-oriented programs, such as flow-insensitive points-to analyses for Java, which typically use finite (non-numerical) domains and aggressive abstraction strategies. On the other hand, our observation comes from the interval analysis for C programs, which uses an expressive infinite abstract domain.

About the remaining alarms. Although we have reduced the number of false alarms by 24%, most of the remaining alarms are still spurious. The remaining alarms arise from various reasons, e.g., imprecise heap abstraction, string abstraction, handling of external library functions, widening at loops, etc. Reducing them is another challenge and beyond the scope of this work. For instance, we observed several cases where the precision improvement by context-sensitivity was not sufficient to remove buffer-overrun alarms. For example, in `grep-2.5`, we observed that some buffer accesses are of the following pattern:

```
void addlists (char **new) {
  for (i=0; new[i] != NULL; i++) { ... }
}
```

Our technique analyzed procedure `addlists` separately for its calling contexts, and hence it accurately inferred the size of buffer `new`. In this case, however, the improved accuracy for `new` is not sufficient to prove the safety of the underlined buffer access, since the analysis cannot analyze the loop accurately (variable `i` has $[0, +\infty]$ inside the loop). In order to prove `i` $<$ `sizeof(new)`, we need to analyze array elements separately and accurately

The presence of many false alarms do not always mean that the analysis is practically useless. In practice, not all of those alarms are given to the end users. The final alarm reports are given in a way to minimize the alarm investigation burdens. For instance, we use alarm the clustering [Lee et al. 2012] and statistical ranking techniques [Jung et al. 2005] to reduce the number of final alarms (by 45% via alarm clustering) and to filter out highly probable false alarms.

Comparison with the k-callstrings approach. According to our experience, the k-callstrings approach did not scale when it is used with the interval abstract domain for analyzing C programs. The 2- and 3-callstrings approaches did not stop after 30 minutes for programs over 10KLOC. Even the 1-callstrings approach was slow and did not scale over 40KLOC. For instance, the 3-callstrings approach succeeded to analyze `spell-1.0` in 11.9s (with 30 alarms reported), it did not stop for `bc-1.06`. We also noticed that even for small programs, the 1- and 2-callstrings approaches were not cost-effective. In spell-1.0, the 1-callstrings approach reported 36 alarms in 2.1s, the 2-callstrings 35 alarms in 7.1s, and the 3-callstrings 30 alarms in 11.9s. On the other hand, our analysis reports 30 alarms in 0.9s.

7.3. Selective Octagon Analysis

We have implemented our selective relational analysis (Section 6.1) on top of the octagon-analysis version of our baseline analyzer. We compare the performance of our selective analysis with an existing octagon analysis based on the syntactic variable packing [Miné 2006; Oh et al. 2012]. The syntactic packing approach relates variables together if they are involved in the same syntactic block [Miné 2006]. We limited the maximum pack size by 10 in the syntactic packing strategy, since otherwise the analysis did not scale.

Program	**LOC**	**#Variable**	**#Query**	**Syntactic Packing Approach**			
				proven	**time**	**mem**	**pack**
calculator-1.0	298	197	10	2	0.3	63	18 (7.3)
spell-1.0	2,213	531	16	1	4.8	109	119 (7.7)
barcode-0.96	4,460	2,002	37	16	11.8	221	276 (8.1)
httptunnel-3.3	6,174	1,908	28	16	26.0	220	454 (7.0)
bc-1.06	13,093	2,194	10	2	247.1	945	606 (7.8)
tar-1.17	20,258	5,332	17	7	1,043.2	1,311	1,259 (7.5)
less-382	23,822	4,482	13	0	3,031.5	1,439	1,017 (6.3)
a2ps-4.14	64,590	16,531	11	0	29,479.3	2,304	2,608 (7.8)
Total	135,008	33,177	142	44	33,840.3	6,611	

Program	**Our Selective Relational Analysis**						**Comparison**	
	proven	**pre**	**main**	**total**	**mem**	**pack**	**Precision**	**Time**
calculator-1.0	10	0.1	0.1	0.2	52	3 (3.6)	+8	-33.3%
spell-1.0	16	1.7	0.7	2.4	63	6 (11.0)	+15	-50.0%
barcode-0.96	37	12.2	18.3	30.5	100	12 (25.0)	+21	158.5%
httptunnel-3.3	26	10.8	4.5	15.3	105	8 (5.8)	+10	-41.2%
bc-1.06	9	82.3	35.0	117.3	212	4 (4.0)	+7	-52.5%
tar-1.17	17	598.5	63.3	661.8	384	7 (3.9)	+10	-36.6%
less-382	13	2,253.2	596.2	2,849.4	955	8 (6.3)	+13	-6.0%
a2ps-4.14	11	2,223.5	518.2	2,741.7	909	6 (6.7)	+11	-90.7%
Total	139	5,182.3	1,236.3	6,418.6	2,780		+95	-81.0%

Table II: Performance comparison between an octagon analysis with an existing syntactic packing strategy and our selective relational analysis. **#Variable** denotes the number of variables (abstract locations) in the program. **#Query** denotes the number of buffer-overrun queries whose proofs require relational reasoning. **proven** reports the number of queries that are proven by each octagon analysis. **mem** reports the peak memory consumption in megabytes. Each X (Y) in column **pack** represents the number of non-singleton packs (X) and the average size (Y) of the packs used in each relational analysis. **Precision** and **Time** shows additionally proven queries and time consumption by our selective relational analysis compared to the syntactic packing approach.

Table II shows our benchmark programs. Most of the programs are open-source GNU programs. On 15 of those programs, we were able to run our octagon analysis. For these 15 programs, we ran the baseline non-relational analysis, manually inspected alarms reported by the analysis, and identified a subset of these 15 programs that require a relational analysis to prove the absence of some buffer overrun queries. (Some programs do not need a relational analysis at all because none of the alarms for them can be removed just by using a relational abstract domain. [Miné 2006; Farzan and Kincaid 2012]) For these identified programs, we compared the performance of syntactic variable packing with our technique. Regarding queries, we used, as queries, the buffer overrun expressions that could not be proved safe by our baseline (non-relational) analysis but can be proved safe by a relational analysis. We identified such expressions manually. Column **#Query** shows the number of such relational queries that we consider in our experiments. In the experiments, we manually inlined the functions that are involved in the proofs of the target queries, so that our selective relational analysis and the syntactic packing approach are run under context-sensitivity.

The results show that our selective octagon analysis has a competitive precision-cost balance. Among 142 queries in total, our analysis is able to prove 139 (97.9%) queries in 1,236.3s. On the other hand, the octagon analysis with syntactic packing proved 44 (32.6%) queries in 33,840.3s; the syntactic packing heuristic often fails to prescribe variable relationships necessary to prove queries. Our analysis is even faster than the counterpart in most cases because it selectively turns on relational analysis. In the experiments, all the queries proved by the syntactic packing method were also proved by our semantic packing technique.

One thing to note is that running our pre-analysis is feasible in practice even though it is fully relational. The bottlenecks of a fully relational octagon analysis are the mem-

Program	Flow-Insensitive		Selective Flow-Sensitive				Flow-Sensitive	
	time	alarm	pre	main	total	alarm	time	alarm
spell-1.0	0.2	37	0.5	0.24	0.8	35	1.3	35
barcode-0.96	0.8	460	3.6	1.04	4.7	434	13.5	434
bc-1.06	1.7	648	43.3	11.63	54.9	616	137.4	614
tar-1.17	2.8	1,042	20.2	15.14	35.3	832	88.8	812
less-382	2.4	727	31.8	45.39	77.2	683	217.1	683
parser	3.4	2,196	23.2	17.13	40.3	2,138	80.8	2,136
sed-4.0.8	5.2	938	36.7	5.88	42.6	842	95.4	842
grep-2.5	1.6	969	8.0	3.03	11.0	914	29.1	914
make-3.76	3.1	2,153	28.0	35.44	63.5	1,438	144.7	1,262
bison-2.5	11.2	1,717	139.7	27.01	166.7	1,445	309.3	1,436
TOTAL	32.3	10,887	335.0	161.93	496.9	9,377	1117.2	9,168

Table III: Experimental results of select flow-sensitive analysis. **pre** means the time spent during the impact pre-analysis and **main** denotes the time for the selective main analysis. All **time**s are in seconds.

ory costs for representing $2|\mathsf{Var}| \times 2|\mathsf{Var}|$ matrices and the expensive strong closure operation [Miné 2006] whose time complexity is cubic in the number of variables. Thanks to the simplicity of the abstract domain ($\bigstar$ or $\top$), we can reduce the memory cost using a sparse representation for the matrices. For the closure operation, we use Dijkstra's algorithm and compute the shortest-path closure [Miné 2006] instead of the strong closure. In our experiments, using the shortest-path closure made no difference in the pre-analysis precision.

7.4. Selective Flow-Sensitive Analysis

We have implemented the selective flow-sensitive method (Section 6.2) on top of the interval-domain based SPARROW. For the impact pre-analysis, we used the abstract domain of $\{\bigstar, \top\}$, where $\bigstar$ means the set of non-negative intervals and $\top$ means all intervals. We compared the performance of our selective flow-sensitive analysis with the fully flow-insensitive and flow-sensitive analyses. Table III shows the results.[5]

The results show that our selective method is also effective for finding an appropriate level of flow-sensitivity. In total, the completely flow-insensitive analysis reports 10887 alarms while taking 32.3s. The completely flow-sensitive analysis increases the analysis time by 35 times while reporting 9168 alarms. Our selective flow-sensitive analysis reduces the analysis time of flow-sensitivity from 1117s to 496s (56% reduction) and in doing so it reports 209 (2% increase) more alarms.

One might conjecture that randomly selecting flow-sensitivity targets would also lead to a similar performance. According to our experience, this is definitely not true. For instance, in our experiments with tar-1.13, a random flow-sensitivity that randomly selects half of the program points reports 1,024 alarms, while our selective flow-sensitivity reports 832 alarms.

8. RELATED WORK

Context-Sensitive Analysis. Most of the previous callstrings-based context-sensitive analysis techniques assign contexts to calls in a uniform manner. The k-callstring approach (or k-CFA) [Sharir and Pnueli 1981; Shivers 1991] and its flexible variants [Harrison III 1989], k-object sensitivity [Milanova et al. 2002], and type sensitivity [Smaragdakis et al. 2011] are such cases. All these techniques generate calling contexts according to a single fixed policy and do not explore how to tune their parameters (for example, different k values at each call site) for target queries. The hybrid context-sensitivity [Kastrinis and Smaragdakis 2013], which employs multiple poli-

[5]Some information in Table III is different from Table I because the experiments in Table III were conducted using a later version (which were developed from scratch) of SPARROW.

cies of assigning contexts in a single analysis, still does not tailor those policies to the program to analyze.

Although the functional approaches to context-sensitivity [Sharir and Pnueli 1981; Reps et al. 1995; Padhye and Khedker 2013] produce necessary calling-contexts only, they are not applicable to analyses with infinite domains as ours. Functional approaches use abstract values as context instead of callstrings, maintaining separate analysis results of a procedure for each different input abstract state. For instance, Padhey and Khedker [Padhye and Khedker 2013] present a general framework for such value-based context-sensitive analyses, which also produces the exact same four contexts when applied to the example program in Section 2. However, these approaches are only applicable to analyses with finite abstract domains, and hence cannot be used in, for instance, interval analysis.

Refinement-based Analysis. While refinement-based analyses [Plevyak and Chien 1994; Guyer and Lin 2003b; Sridharan and Bodík 2006] are similar to our approach (in that they use a "pre-analysis" to adjust the main analysis precision), there is a fundamental difference in their techniques. Refinement-based approaches (e.g., client-driven analysis [Guyer and Lin 2003b]) start with an *imprecise* analysis and refines the abstraction in response to client queries. On the other hand, our approach starts with a pre-analysis that estimates the impact of the *most precise* main analysis. As a result, our approach provides a precision guarantee, which does not hold in the refinement-based techniques. Furthermore, the principle behind our approach is general; it is applicable to a range of static analyses (such as interval and octagon analyses) with various precision axes (such as context-sensitivity and relational analysis). Existing refinement-based analyses have been special for pointer analyses [Plevyak and Chien 1994; Guyer and Lin 2003b; Sridharan and Bodík 2006].

Unfortunately, it is difficult to directly compare our work with the existing refinement-based techniques. For instance, one approach most similar to ours is the client-driven pointer analysis [Guyer and Lin 2003b], where a main analysis with the most imprecise setting identifies places where higher precision is needed. However, note that the technique has been developed for a particular class of analyses in mind (i.e., pointer analyses). It is unclear how to generalize the technique to, for instance, the kinds of program analyses studied in our paper (i.e., interval analysis and octagon analysis).

Demand-driven Analysis. Our approach is orthogonal to demand-driven analyses [Heintze and Tardieu 2001; Sridharan et al. 2005; Sridharan and Bodík 2006]. While demand-driven analyses aim to reduce analysis costs by computing only the partial solution necessary to answer given queries, we compute the exhaustive solution with an abstraction tailored to the queries (our analysis is run once for the entire set of queries). Both approach can complement each other.

Inference of Analysis Parameters. In a high level, our approach suggests a novel technique for *analysis-parameter inference* [Liang et al. 2011; Naik et al. 2012; Zhang et al. 2013; Zhang et al. 2014]. There are many parameters to tune in static analysis, to improve either precision or scalability. Finding a parameter setting with which the analysis improves the precision without significant slowdown has been a central problem in static analysis research. Liang et al. [Liang et al. 2011] use machine learning to find a minimal abstraction, an abstraction that is minimal yet sufficient to prove all the queries provable by the most precise abstraction, for given queries. Guided by the number of queries each analysis run has proven, the machine learning algorithm infers a minimal abstraction (i.e., the k value for each function in context-sensitivity). However, they study the minimal abstraction itself and provide no practical solutions for selective context-sensitivity. Zhang et al. [Zhang et al. 2013] present a technique for finding the optimum abstraction, a cheapest abstraction that proves the query, but it is applicable only to disjunctive analyses. Naik et al. [Naik et al. 2012] use a dynamic analysis to select an appropriate parameter for a given query, which is guaranteed to be a necessary condition to prove the query. Zhang et al. [Zhang et al. 2014] presents a technique for finding a good program abstraction of program analyses written in Datalog. The proposed technique uses a counterexample-guided abstraction refinement, where the technique guarantees that a failed run excludes all of the abstractions that lead to similar failures. Our work provides a new contribution to this line of research, where we use a static pre-analysis to infer an analysis parameter. Our technique is also applicable to non-disjunctive analyses and non-datalog-based analyses.

Existing Relational Analyses. Our selective octagon analysis is similar to the existing octagon analyses (such as [Miné 2006; Oh et al. 2012; Farzan and Kincaid 2012]) in that they use variable packing and and hence they are partially relational. However, while we selectively construct variable packs that likely benefit the final analysis precision, existing analyses construct variable packs based on syntactic heuristics [Miné 2006; Oh et al. 2012] or program dependencies [Farzan and Kincaid 2012].

9. CONCLUSION

9.1. Summary

We proposed a method of designing a selective "X-sensitive" analysis, where the selection is guided by an impact pre-analysis. We followed this approach, presented three program analyses that selectively apply precision-improving techniques, and demonstrated their effectiveness with experiments in a realistic setting. The first was a selective context-sensitive analysis that receives guidance from an impact pre-analysis. The experiments with realistic benchmarks showed that the method reduces the number of false alarms of the context-insensitive interval analysis by 24.4%, while increasing the analysis cost by 27.8%. The second example was a selective relational analysis with octagons using the same idea of impact pre-analysis. The experiments showed that our selective octagon analysis proves 95 more queries than the existing one based on the syntactic variable packing does, and reduces the analysis cost by 81%. The last example was a selective flow-sensitive analysis with intervals, where the results showed that our method reduces the cost of the flow-sensitive analysis by 56%, while reporting 2% more false alarms. We believe that our approach can be used for developing other selective analyses as well, e.g., selective path-sensitive analysis, selective loop-unrolling, etc.

9.2. Open Issues

(1) *Application to other types of client analyses*: In this paper, we have demonstrated that our technique is effective for context-sensitivity, flow-sensitivity, and relational analysis with a buffer-overrun client analysis. It remains an open question whether our technique can be effectively applicable to other types of client analyses (e.g., resource leaks, race detection, etc) or not.

(2) *Performance improvement of selective flow-sensitivity*: Although our selective flow-sensitive analysis reduced the analysis time of full flow-sensitivity by 56%, the current technique for flow-sensitivity requires relatively high costs. Developing a new technique for achieving a better cost/accuracy balance would be an interesting future work. For example, we could design a new impact pre-analysis domain that shows better performance than our simple $\top - \bigstar$ domain.

(3) *Combining "sensitivities"*: Although we presented three selective program analyses separately, we do not address the problem of combining those analyses. How could

we design a pre-analysis that estimates context-sensitivity, flow-sensitivity, and relational constraints worth analyzing for at the same time? Naively designing an impact pre-analysis that is maximally precise in all three precision axes would yield an intractable one. We think that it is an interesting open problem to design an effective pre-analysis for such a combined analysis.

ACKNOWLEDGMENTS

This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning(MSIP) / National Research Foundation of Korea(NRF) (Grant NRF-2008-0062609), and by Samsung Electronics Software Center. This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. B0101-15-0557, Resilient Cyber-Physical Systems Research) and (No.R0190-15-2011,Development of Vulnerability Discovery Technologies for IoT Software Security). Yang was supported by EPSRC.

REFERENCES

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 238–252.

COUSOT, P. AND COUSOT, R. 1992. Abstract interpretation frameworks. *J. Log. Comput. 2,* 4, 511–547.

DEUTSCH, A. 1997. On the complexity of escape analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '97. ACM, New York, NY, USA, 358–371.

FARZAN, A. AND KINCAID, Z. 2012. Verification of parameterized concurrent programs by modular reasoning about data and control. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

GUYER, S. AND LIN, C. 2003a. Client-driven pointer analysis. In *Proceedings of the International Symposium on Static Analysis*. 214–236.

GUYER, S. Z. AND LIN, C. 2003b. Client-driven pointer analysis. In *Proceedings of Static Analysis Symposium*.

HARRISON III, W. L. 1989. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*.

HEINTZE, N. AND TARDIEU, O. 2001. Demand-driven pointer analysis. In *Proceedings of The ACM SIGPLAN Conference on Programming Language Design and Implementation*.

JUNG, Y., KIM, J., SHIN, J., AND YI, K. 2005. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *Proceedings of the International Symposium on Static Analysis*. 203–217.

KASTRINIS, G. AND SMARAGDAKIS, Y. 2013. Hybrid context-sensitivity for points-to analysis. In *Proceedings of The ACM SIGPLAN Conference on Programming Language Design and Implementation*.

LEE, W., LEE, W., AND YI, K. 2012. Sound non-statistical clustering of static analysis alarms. In *VMCAI 2012: 13th International Conference on Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science Series, vol. 7148. Springer, 299–314.

LIANG, P., TRIPP, O., AND NAIK, M. 2011. Learning minimal abstractions. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

MILANOVA, A., ROUNTEV, A., AND RYDER, B. G. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*.

MINÉ, A. 2006. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation 19,* 1, 31–100.

NAIK, M., YANG, H., CASTELNUOVO, G., AND SAGIV, M. 2012. Abstractions from tests. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

OH, H., , LEE, W., HEO, K., YANG, H., AND YI, K. 2014. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of The ACM SIGPLAN Conference on Programming Language Design and Implementation*.

OH, H., HEO, K., LEE, W., LEE, W., AND YI, K. Sparrow. `http://ropas.snu.ac.kr/sparrow`.

OH, H., HEO, K., LEE, W., LEE, W., AND YI, K. 2012. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of The ACM SIGPLAN Conference on Programming Language Design and Implementation*.

OH, H. AND YI, K. 2010. An algorithmic mitigation of large spurious interprocedural cycles in static analysis. *Software: Practice and Experience 40,* 8, 585–603.

OH, H. AND YI, K. 2013. Access-based abstract memory localization in static analysis. *Science of Computer Programming 78,* 9, 1701–1727.

PADHYE, R. AND KHEDKER, U. P. 2013. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*. SOAP '13. ACM, New York, NY, USA, 31–36.

PLEVYAK, J. AND CHIEN, A. A. 1994. Precise concrete type inference for object-oriented languages. In *Proceedings of The ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 49–61.

RIVAL, X. AND MAUBORGNE, L. 2007. The trace partitioning abstract domain. *ACM Trans on Programming Languages and System 29,* 5, 26–51.

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 189–234.

SHIVERS, O. G. 1991. Control-flow analysis of higher-order languages -or- taming lambda. Ph.D. thesis, CMU.

SMARAGDAKIS, Y., BRAVENBOER, M., AND LHOTÁK, O. 2011. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

SRIDHARAN, M. AND BODÍK, R. 2006. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of The ACM SIGPLAN Conference on Programming Language Design and Implementation*.

SRIDHARAN, M., GOPAN, D., SHAN, L., AND BODÍK, R. 2005. Demand-driven points-to analysis for Java. In *Proceedings of The ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*.

ZHANG, X., MANGAL, R., GRIGORE, R., NAIK, M., AND YANG, H. 2014. On abstraction refinement for program analyses in datalog. In *Proceedings of The ACM SIGPLAN Conference on Programming Language Design and Implementation*.

ZHANG, X., NAIK, M., AND YANG, H. 2013. Finding optimum abstractions in parametric dataflow analysis. In *Proceedings of The ACM SIGPLAN Conference on Programming Language Design and Implementation*.

A. PROOFS

In this appendix, we prove the impact guarantee (Proposition 5.17) of our selective context-sensitive analysis as well as the correctness of our reachability-based pre-analysis algorithm (Lemma 5.7).

Remark A.1. In the rest of this appendix, we generalize the notion of the impact pre-analysis, and write $\mathsf{PA}_K : \mathbb{C} \to \mathbb{S}^\sharp$ for the solution of the pre-analysis under context-sensitivity K. In the body of this paper, we have discussed our pre-analysis and its reachability-based algorithm only under full context-sensitivity ($K = K_\infty$). However, the correctness of our pre-analysis (Lemma 5.7) holds with arbitrary context selector K, as its proof does not assume a particular instance of K (we prove this generalized lemma in A.2). This implies that, regardless of the underlying context-sensitivity, we can compute the least solution of the pre-analysis (7) via our reachability algorithm.

A.1. Proof of Proposition 5.17

PROOF. To show:

$$\forall \kappa \in \mathsf{K}(\mathsf{fid}(c)).\ \mathsf{MA}_{\mathsf{K}}(\kappa, c) \in \gamma(\top[x \mapsto \mathsf{PA}_{K_\infty}(c)(x)])$$

It is proved by Lemma A.2 and A.6:

$$\begin{array}{lll} & \gamma(\top[x \mapsto \mathsf{PA}_{K_\infty}(c)(x)]) & \\ = & \gamma(\top[x \mapsto \mathsf{PA}_{\mathsf{K}}(c)(x)]) & \textbf{(Lemma A.2)} \\ \supseteq & \gamma(\mathsf{PA}_{\mathsf{K}}(c)) & \textbf{(Soundness of } \top\textbf{)} \\ \ni & \mathsf{MA}_{\mathsf{K}}(\kappa, c) & \textbf{(Lemma A.6)} \end{array}$$

□

What is non-trivial is the first equality (Lemma A.2), which asserts that the result of the pre-analysis under full context-sensitivity coincides with that of the pre-analysis under our selective context-sensitivity K (Definition 5.15), as far as the selected query (c, x) is concerned.

LEMMA A.2 (PRE-ANALYSIS COINCIDENCE). *Let (c_q, x_q) be a query. Let PA_{K_∞} be the pre-analysis result with full context-sensitivity. Let K be the selective context-sensitivity for query (c_q, x_q) defined using PA_{K_∞} (Definition 5.15). Let PA_{K} be the result of the pre-analysis under the selective context-sensitivity K. Then, $\mathsf{PA}_{K_\infty}(c_q) = \mathsf{PA}_{\mathsf{K}}(c_q) = \bot$ or*

$$\mathsf{PA}_{K_\infty}(c_q)(x_q) = \mathsf{PA}_{\mathsf{K}}(c_q)(x_q).$$

PROOF. It is sufficient to show that, in the value-flow graph, the query (c_q, x_q) is reachable from a source (c_0, x_0) under the full context-sensitivity if and only if (c_q, x_q) is reachable from (c_0, x_0) under the selective context-sensitivity K:

$$\begin{array}{l} \forall (c_0, x_0) \in \Phi. \\ \qquad (c_0, x_0) \hookrightarrow^{\dagger}_{K_\infty} (c_q, x_q) \iff (c_0, x_0) \hookrightarrow^{\dagger}_{\mathsf{K}} (c_q, x_q). \end{array}$$

— ($\Longrightarrow$) By Lemma A.3.
— ($\Longleftarrow$) When $(c_0, x_0) \in \Phi_{(c_q, x_q)}$, by the definition of $\Phi_{(c_q, x_q)}$. When $(c_0, x_0) \notin \Phi_{(c_q, x_q)}$, by Lemma A.5.

□

We prove this lemma in two steps. We first show that for every $(c, x) \in \Phi_{(c_q, x_q)}$, we have $(c, x) \hookrightarrow^{\dagger}_{\mathsf{K}} (c_q, x_q)$ (Lemma A.3). Then, we prove that there is no $(c', x') \in \Phi \backslash \Phi_{(c_q, x_q)}$ such that $(c', x') \hookrightarrow^{\dagger}_{\mathsf{K}} (c_q, x_q)$ (Lemma A.5). Comprising these two lemmas, we can prove Lemma A.2.

The following lemma shows that, for every K_∞-valid value-flow path from sources to the query, we can always find the corresponding K-valid value-flow path.

LEMMA A.3. *Suppose $(c_0, x_0) \in \Phi_{(c_q, x_q)}$ and consider c_i, κ_i, x_i such that*

$$\begin{array}{l} (\iota, \epsilon) \rightarrow^{*}_{K_\infty} (c_0, \kappa_0) \wedge \\ ((c_0, \kappa_0), x_0) \hookrightarrow_{K_\infty} ((c_1, \kappa_1), x_1) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q, \kappa_q), x_q). \end{array}$$

Then, we have

$$((c_0, \kappa_0'), x_0) \hookrightarrow_{\mathsf{K}} ((c_1, \kappa_1'), x_1) \hookrightarrow_{\mathsf{K}} \cdots \hookrightarrow_{\mathsf{K}} ((c_q, \kappa_q'), x_q).$$

where $\kappa_i' = \kappa_i \ominus \kappa_0$.

Before proving this lemma, we assume that the given program is *well-formed* in the following sense.

Definition A.4 (*Well-formed Programs*). We say a program is well-formed with respect to a query (c_q, x_q) iff for every source $(c_0, x_0) \in \Phi_{(c_q, x_q)}$ and its valid value-flow path

$$((c_0,\kappa_0),x_0) \hookrightarrow_{K_\infty} \cdots ((c_i,\kappa_i),x_i) \cdots \hookrightarrow_{K_\infty} ((c_q,\kappa_q),x_q)$$

the initial context κ_0 does not include any intermediate call site c_i, i.e., $c_i \notin \kappa_0$.

Intuitively, the above condition excludes dependency paths generated by (recursive) cycles in the program.

Now we prove the lemma:

PROOF. By the definition of K,

$$\forall i.\ \kappa'_i = \kappa_i \ominus \kappa_0 \in \mathsf{K}.$$

We show that

$$\begin{array}{l}\forall 0 \le i < n.\\ ((c_i,\kappa_i),x_i) \hookrightarrow_{K_\infty} ((c_{i+1},\kappa_{i+1}),x_{i+1})\\ \implies ((c_i,\kappa'_i),x_i) \hookrightarrow_{\mathsf{K}} ((c_{i+1},\kappa'_{i+1}),x_{i+1})\end{array}$$

where $c_n = c_q$, $\kappa_n = \kappa_q$, and $x_n = x_q$. This simply amounts to showing the following:

$$\begin{array}{l}\forall 0 \le i < n.\\ ((c_i,\kappa_i) \rightarrow_{K_\infty} (c_{i+1},\kappa_{i+1})) \implies ((c_i,\kappa'_i) \rightarrow_{\mathsf{K}} (c_{i+1},\kappa'_{i+1})).\end{array}$$

(1) $c_i \notin \mathbb{C}_c \uplus \mathbb{C}_x$:
By the definition of $\rightarrow_{K_\infty}$,

$$\kappa_i = \kappa_{i+1} \qquad \kappa_i \ominus \kappa_0 = \kappa_{i+1} \ominus \kappa_0.$$

By the definition of $\rightarrow_{\mathsf{K}}$,

$$(c_i, \kappa_i \ominus \kappa_0) \rightarrow_{\mathsf{K}} (c_{i+1}, \kappa_{i+1} \ominus \kappa_0).$$

(2) $c_i \in \mathbb{C}_c$:
By the definition of $\rightarrow_{K_\infty}$,

$$\kappa_{i+1} = c_i \cdot \kappa_i \qquad \kappa_{i+1} \ominus \kappa_0 = (c_i \cdot \kappa_i) \ominus \kappa_0.$$

By the definition of K,

$$(c_i \cdot \kappa_i) \ominus \kappa_0 \in \mathsf{K}.$$

By the Definition A.4,

$$(c_i \cdot \kappa_i) \ominus \kappa_0 \neq \epsilon$$

and hence

$$(c_i \cdot \kappa_i) \ominus \kappa_0 = c_i \cdot (\kappa_i \ominus \kappa_0) \in \mathsf{K}.$$

Therefore, by the definition of $::_{\mathsf{K}}$ and $\rightarrow_{\mathsf{K}}$, we have

$$(c_i, \kappa_i \ominus \kappa_0) \rightarrow_{\mathsf{K}} (c_{i+1}, c_i ::_{\mathsf{K}} (\kappa_i \ominus \kappa_0)).$$

(3) $c_i \in \mathbb{C}_x$:
By the definition of $\rightarrow_{K_\infty}$,

$$\kappa_i = \mathsf{callof}(c_{i+1}) \cdot \kappa_{i+1}$$

and hence

$$\kappa_i \ominus \kappa_0 = (\mathsf{callof}(c_{i+1}) \cdot \kappa_{i+1}) \ominus \kappa_0.$$

Now we split into two cases.
(a) When $(\mathsf{callof}(c_{i+1}) \cdot \kappa_{i+1}) \ominus \kappa_0 \neq \epsilon$,
we have

$$\begin{array}{rl}\kappa'_i = & (\mathsf{callof}(c_{i+1}) \cdot \kappa_{i+1}) \ominus \kappa_0\\ = & \mathsf{callof}(c_{i+1}) \cdot (\kappa_{i+1} \ominus \kappa_0)\\ = & \mathsf{callof}(c_{i+1}) \cdot \kappa'_{i+1}\end{array}$$

From the definition of K, we have $\kappa'_i, \kappa'_{i+1} \in \mathsf{K}$. Therefore, by the definition of $::_{\mathsf{K}}$, we have

$$\kappa'_i = \mathsf{callof}(c_{i+1}) ::_{\mathsf{K}} \kappa'_{i+1}.$$

By the definition of $\rightarrow_{\mathsf{K}}$,

$$(c_i, \kappa'_i) \rightarrow_{\mathsf{K}} (c_{i+1}, \kappa'_{i+1}).$$

(b) When $(\mathsf{callof}(c_{i+1}) \cdot \kappa_{i+1}) \ominus \kappa_0 = \epsilon$, we have

$$\kappa_i \ominus \kappa_0 = \kappa_{i+1} \ominus \kappa_0 = \epsilon$$

which means that $\kappa'_i = \kappa'_{i+1} = \epsilon$. Also, note that we have

$$\mathsf{callof}(c_{i+1}) \in \kappa_0 \tag{18}$$

from $(\mathsf{callof}(c_{i+1}) \cdot \kappa_{i+1}) \ominus \kappa_0 = \epsilon$. If we assume that $\mathsf{callof}(c_{i+1}) \cdot \kappa'_{i+1} \notin \mathsf{K}$, we can conclude that

$$(c_i, \kappa'_i) \rightarrow_{\mathsf{K}} (c_{i+1}, \kappa'_{i+1}).$$

Now we prove that the assumption is actually true:

$$\mathsf{callof}(c_{i+1}) \cdot \kappa'_{i+1} = \mathsf{callof}(c_{i+1}) \notin \mathsf{K}.$$

Suppose $\mathsf{callof}(c_{i+1}) \in \mathsf{K}$. Then, there should exist $c_j = \mathsf{callof}(c_{i+1}) \in \mathbb{C}_c$ such that

$$((c_0, \kappa_0), x_0) \hookrightarrow^*_{K_\infty} ((c_j, \kappa_j), x_j) \hookrightarrow^*_{K_\infty} ((c_n, \kappa_n), x_n)$$

and

$$(c_j \cdot \kappa_j) \ominus \kappa_0 = c_j \in \mathsf{K}.$$

By the Definition A.4, $c_j = \mathsf{callof}(c_{i+1}) \notin \kappa_0$, which contradicts (18).

□

The following lemma formalizes the fact that our selective context-sensitive analysis designed in Section 4 isolates undistinguished contexts from distinguished contexts: if a source does not reach the query in the fully context-sensitive pre-analysis, then the source does not reach the query in the selective context-sensitive pre-analysis as well.

LEMMA A.5 (ISOLATION). *For all* $(c_0, x_0) \in \Phi \setminus \Phi_{(c_q, x_q)}$,

$$(c_0, x_0) \not\hookrightarrow^\dagger_{\mathsf{K}} (c_q, x_q).$$

PROOF. Suppose we have $(c_0, x_0) \in \Phi \setminus \Phi_{(c_q, x_q)}$ such that

$$(c_0, x_0) \hookrightarrow^\dagger_{\mathsf{K}} (c_q, x_q).$$

Then, by the definition of $\hookrightarrow^\dagger_{\mathsf{K}}$, there exists a $\hookrightarrow_{\mathsf{K}}$-path

$$((c_0, \kappa_0), x_0) \hookrightarrow^*_{\mathsf{K}} ((c_q, \kappa_q), x_q)$$

for some κ_0 and κ_q, which means that we have a $\rightarrow_{\mathsf{K}}$-path

$$(c_0, \kappa_0) \rightarrow_{\mathsf{K}} \cdots \rightarrow_{\mathsf{K}} (c_q, \kappa_q). \tag{19}$$

Because $\kappa_q \in \mathsf{K}(\mathsf{fid}(c_q))$, by the definition of K we have a $\hookrightarrow_{K_\infty}$-path from some source $(c_s, x_s) \in \Phi_{(c_q, x_q)}$

$$((c_s, \kappa_s), x_s) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q, \kappa'_q), x_q),$$

where $\kappa_q = \kappa'_q \ominus \kappa_s$. Then, by Lemma A.3, we have a $\hookrightarrow_{\mathsf{K}}$-path

$$((c_s, \epsilon), x_s) \rightarrow_{\mathsf{K}} \cdots \hookrightarrow_{\mathsf{K}} ((c_q, \kappa_q), x_q) \tag{20}$$

from which we can derive a $\rightarrow_{\mathsf{K}}$-path

$$(c_s, \epsilon) \rightarrow_{\mathsf{K}} \cdots \rightarrow_{\mathsf{K}} (c_q, \kappa_q). \tag{21}$$

This path should be distinct from the path (19). Let (c_i, κ_i) be the farthest point from the query that (19) and (21) agree. We further assume that we have chosen the path (20) such that among all the $\rightarrow_{\mathsf{K}}$-paths from (c_s, ϵ) to (c_q, κ_q), (21) has the longest common suffix with (19). Let (c_{i-1}, κ_{i-1}) and $(c'_{i-1}, \kappa'_{i-1})$ be the first diverging point from the query such that

$$\begin{aligned}(c_0, \kappa_0) \rightarrow_{\mathsf{K}} \cdots \rightarrow_{\mathsf{K}} (c_{i-1}, \kappa_{i-1})&\\ \rightarrow_{\mathsf{K}} (c_i, \kappa_i) \rightarrow_{\mathsf{K}} \cdots (c_q, \kappa_q)&\end{aligned} \tag{22}$$

and

$$\begin{aligned}(c_s, \epsilon) \rightarrow_{\mathsf{K}} \cdots \rightarrow_{\mathsf{K}} (c''_{i-1}, \kappa''_{i-1})&\\ \rightarrow_{\mathsf{K}} (c_i, \kappa_i) \rightarrow_{\mathsf{K}} \cdots (c_q, \kappa_q)&\end{aligned} \tag{23}$$

where $(c_{i-1}, \kappa_{i-1}) \neq (c''_{i-1}, \kappa''_{i-1})$. Note that (19) and (21) agree each other at least at the query point. We now show that this is not possible.

(1) When $c_i \notin \mathbb{C}_e \uplus \mathbb{C}_r$:
By the definition of $\rightarrow_K$, we have

$$\kappa_{i-1} = \kappa_i = \kappa''_{i-1}.$$

Thus, we should have

$$c_{i-1} \rightarrow c_i \;\wedge\; c''_{i-1} \rightarrow c_i$$

where $c_{i-1} \neq c'_{i-1}$, which basically means that c_i is a join point and c_{i-1} and c''_{i-1} exercise different branches. However, because we have considered all possible valid paths from sources in $\Phi_{(c_q, x_q)}$ to (c_q, x_q), we always find another path

$$\begin{aligned}(c_s, \epsilon) \rightarrow_{\mathsf{K}} \cdots \rightarrow_{\mathsf{K}} (c_{i-1}, \kappa''_{i-1})&\\ \rightarrow_{\mathsf{K}} (c_i, \kappa_i) \rightarrow_{\mathsf{K}} \cdots (c_q, \kappa_q)&\end{aligned}$$

whenever we have path (23). Therefore, (c_i, κ_i) cannot be the farthest point.

(2) When $c_i \in \mathbb{C}_e$:
By the definition of $\rightarrow_{\mathsf{K}}$,

$$c_{i-1} ::_{\mathsf{K}} \kappa_{i-1} = \kappa_i = c''_{i-1} ::_{\mathsf{K}} \kappa''_{i-1}.$$

It should be either $\kappa_i = \epsilon$ and $c_{i-1} \cdot \kappa_{i-1}, c''_{i-1} \cdot \kappa''_{i-1} \notin \mathsf{K}$, or $\kappa_i \neq \epsilon$ and $(c_{i-1}, \kappa_{i-1}) = (c''_{i-1}, \kappa''_{i-1})$. From (21) and the definition of K, we have

$$\kappa_i = \kappa'_i \ominus \kappa_s = (c''_{i-1} \cdot \kappa'_{i-1}) \ominus \kappa_s$$

for some κ'_i and κ'_{i-1} such that

$$(c_s, \kappa_s) \rightarrow \cdots \rightarrow (c''_{i-1}, \kappa'_{i-1}) \rightarrow (c_i, \kappa'_i) \rightarrow (c_q, \kappa'_q).$$

From the Definition A.4, we have

$$(c''_{i-1} \cdot \kappa'_{i-1}) \ominus \kappa_s \neq \epsilon.$$

We can deduce from this $(c_{i-1}, \kappa_{i-1}) = (c''_{i-1}, \kappa''_{i-1})$. Therefore, (c_i, κ_i) cannot be the farthest point.

(3) When $c_i \in \mathbb{C}_r$:
By the definition of $\rightarrow_{\mathsf{K}}$,

$$\kappa_{i-1} = \mathsf{callof}(c_i) ::_{\mathsf{K}} \kappa_i = \kappa''_{i-1}.$$

Also, by the definition of return edge $\dashrightarrow$, we can deduce $c_{i-1} = c''_{i-1}$ from $c_{i-1} \dashrightarrow c_i$ and $c''_{i-1} \dashrightarrow c_i$. Therefore, (c_i, κ_i) cannot be the farthest point.

□

The following lemma shows that our pre-analysis algorithm correctly estimates the behavior of the main analysis if they use the same context selector.

LEMMA A.6. *Let K be an arbitrary context selector. Let $\mathsf{MA}_K \in \mathbb{D}$ be the main analysis result, i.e., a solution of* (6)*, under the K. Let $\mathsf{PA}_K \in \mathbb{C} \to \mathbb{S}^\sharp$ be the result of the reachability-based algorithm (Definition 5.6) under the K. Then,*

$$\forall c \in \mathbb{C}, \kappa \in \mathbb{C}_c^*.\ \mathsf{MA}_K(c, \kappa) \in \gamma(\mathsf{PA}_K(c)).$$

PROOF. This lemma is proved by Lemma 5.1 and Lemma 5.7, where the proof of Lemma 5.1 is immediate from the abstract interpretation framework [Cousot and Cousot 1977; 1992] and we omit the proof. We prove Lemma 5.7 in A.2. □

A.2. Proof of Lemma 5.7

Let PA_K be the result of our pre-analysis under context-sensitivity K. We show that PA_K is equivalent to the least such X of (7) when the underlying context selector is K.

To show the equivalence, we first define a new graph and use it to construct an element $\mathcal{X} \in \mathbb{C}_K \to \mathbb{S}^\sharp$ based on the reachability over this graph. Then, we prove that $\mathcal{X}$ is the least solution of (7) (Lemmas A.7 and A.8) and PA_K is equivalent to $\mathcal{X}$ (Lemma A.9).

In the below, we spell out the details of constructing $\mathcal{X}$:

(1) We define a context-enriched value-flow graph $(\Omega, \hookrightarrow_K)$ with the node set $\Omega = \mathbb{C}_K \times \mathsf{Var}$ and the edge set $(\hookrightarrow_K) \subseteq \Omega \times \Omega$ in Definition 5.4.
(2) Let V be the set of (c, κ)'s reachable from (ι, ϵ):
$$V = \{(c, \kappa) \mid (\iota, \epsilon) \rightarrow_K^* (c, \kappa)\}$$
(3) We define a set Ω_v of generators for each abstract value $v \in \mathbb{V}$:
$$\begin{aligned}\Omega_v = {} & (\textit{if}\ (v = \top_v)\ \textit{then}\ \{((\iota, \epsilon), x) \mid x \in \mathsf{Var}\}\ \textit{else}\ \{\}) \\ & \cup \{((c, \kappa), x) \mid \mathsf{cmd}(c) = x := e \wedge \mathsf{const}(e) = v\}\end{aligned}$$
(4) Finally, using what we have defined so far, we construct $\mathcal{X} \in \mathbb{D}^\sharp = \mathbb{C}_K \to \mathbb{S}^\sharp$:
$$\begin{aligned}\mathcal{X}(c, \kappa) = {} & \textit{if}\ ((c, \kappa) \notin V)\ \textit{then}\ \bot \\ & \textit{else}\ \lambda x. \bigsqcup\{v \in \mathbb{V} \mid \exists ((c_0, \kappa_0), x_0) \in \Omega_v. \\ & \qquad (c_0, \kappa_0) \in V\ \wedge \\ & \qquad ((c_0, \kappa_0), x_0) \hookrightarrow_K^* ((c, \kappa), x)\}\end{aligned}$$

LEMMA A.7. *The $\mathcal{X}$ is a solution of* (7)*. That is,*

$$s_I^\sharp \sqsubseteq \mathcal{X}(\iota, \epsilon)\ \wedge\ F^\sharp(\mathcal{X}) \sqsubseteq \mathcal{X}.$$

PROOF. The first condition holds because, for all x, $((\iota, \epsilon), x)$ belongs to $\Omega_{\top_v}$ and $(\iota, \epsilon) \in V$. Hence $\mathcal{X}(\iota, \epsilon) = (\lambda x. \top_v) = \top$.

Next we show that

$$\forall (c, \kappa) \in \mathbb{C}_K.\ F^\sharp(\mathcal{X})(c, \kappa) \sqsubseteq \mathcal{X}(c, \kappa).$$

Pick $(c, \kappa) \in \mathbb{C}_K$. Suppose that

$$(c, \kappa) \notin V.$$

Then, $\mathcal{X}(c,\kappa) = \bot$ by the definition of $\mathcal{X}$. Also, for every $(c_0,\kappa_0) \in \mathbb{C}_K$, if $(c_0,\kappa_0) \to_K (c,\kappa)$, then $(c_0,\kappa_0) \notin V$, which implies that

$$\mathcal{X}(c_0,\kappa_0) = \bot.$$

Using these observations, we derive the desired relationship as follows:

$$\begin{aligned} F^\sharp(\mathcal{X})(c,\kappa) &= [\![\mathsf{cmd}(c)]\!](\bigsqcup\{\mathcal{X}(c_0,\kappa_0) \mid (c_0,\kappa_0) \to_K (\iota,\kappa)\}) \\ &= [\![\mathsf{cmd}(c)]\!](\bot) \\ &= \bot \\ &= \mathcal{X}(\iota,\kappa). \end{aligned}$$

The third equality holds because $[\![cmd]\!](\bot) = \bot$ for every cmd.

Let us now consider the case that

$$(c,\kappa) \in V.$$

In this case,

$$\mathcal{X}(c,\kappa) \neq \bot.$$

If $F^\sharp(\mathcal{X})(c,\kappa) = \bot$, the desired relationship follows immediately from the fact that $\bot$ is the least abstract state. Suppose

$$F^\sharp(\mathcal{X})(c,\kappa) \neq \bot.$$

We need to show that

$$\forall x \in \mathsf{Var}.\ F^\sharp(\mathcal{X})(c,\kappa)(x) \sqsubseteq \mathcal{X}(c,\kappa)(x).$$

Pick $x \in \mathsf{Var}$. Let $v = \mathcal{X}(c,\kappa)(x)$. Also, define w to be $\mathsf{const}(e)$ if $\mathsf{cmd}(c)$ is a command of the form $x := e$ for some expression e; otherwise, let $w = \bot_v$. By the definition of $\mathcal{X}$,

$$\begin{aligned} &\forall v'.\ \forall((c_0,\kappa_0),x_0) \in \Omega_{v'}. \\ &\qquad ((c_0,\kappa_0) \in V \ \wedge\ ((c_0,\kappa_0),x_0) \hookrightarrow_K^* ((c,\kappa),x)) \\ &\qquad\qquad \implies v' \sqsubseteq v. \end{aligned}$$

This implies two important facts. First,

$$w \sqsubseteq v \tag{24}$$

because $w = \bot_v$, or $((c,\kappa),x) \in \Omega_w$ and $(c,\kappa) \in V$. Second,

$$\begin{aligned} &\forall((c_0,\kappa_0),x_0). \\ &\qquad ((c_0,\kappa_0) \in V \ \wedge\ ((c_0,\kappa_0),x_0) \hookrightarrow_K ((c,\kappa),x)) \\ &\qquad\qquad \implies \mathcal{X}(c_0,\kappa_0)(x_0) \sqsubseteq v. \end{aligned} \tag{25}$$

Meanwhile, by the definitions of $F^\sharp$, $\hookrightarrow_K$, and the abstract semantics of primitive commands,

$$\begin{aligned} &F^\sharp(\mathcal{X})(c,\kappa)(x) = \\ &\quad w \sqcup \bigsqcup\{\mathcal{X}(c_0,\kappa_0)(x_0) \mid (c_0,\kappa_0) \in V \\ &\qquad\qquad \wedge ((c_0,\kappa_0),x_0) \hookrightarrow_K ((c,\kappa),x)\}. \end{aligned}$$

Hence, from the two facts in (24) and (25) follows that

$$F^\sharp(\mathcal{X})(c,\kappa)(x) \sqsubseteq \mathcal{X}(c,\kappa)(x)$$

as desired. □

LEMMA A.8. *The $\mathcal{X}$ is a lower bound for every solution of* (7). *That is, for every $X \in \mathbb{D}$,*

$$(s_I^\sharp \sqsubseteq X(\iota, \epsilon) \ \wedge \ F^\sharp(X) \sqsubseteq X) \implies \mathcal{X} \sqsubseteq X.$$

PROOF. Consider $X \in \mathbb{D}$ such that

$$s_I^\sharp \sqsubseteq X(\iota, \epsilon) \ \wedge \ F^\sharp(X) \sqsubseteq X.$$

We have to show that

$$\forall (c, \kappa) \in \mathbb{C}_K.\ \mathcal{X}(c, \kappa) \sqsubseteq X(c, \kappa). \tag{26}$$

First, we show that

$$\forall (c, \kappa) \in V.\ X(c, \kappa) \neq \bot. \tag{27}$$

Pick $(c, \kappa) \in V$. By the definition of V,

$$(\iota, \epsilon) \to_K^n (c, \kappa).$$

for some $n \geq 0$. Our proof is by induction on n.

— Base case: $n = 0$ in this case. Hence, $(\iota, \epsilon) = (c, \kappa)$. Since $s_I^\sharp \sqsubseteq X(\iota, \epsilon)$ by assumption,

$$X(\iota, \epsilon) = \top \neq \bot,$$

as desired.

— Inductive case: $n > 0$ in this case. Hence, there exists (c_0, κ_0) such that

$$(\iota, \epsilon) \to_K^{n-1} (c_0, \kappa_0) \to_K (c, \kappa).$$

This implies that $(c_0, \kappa_0) \in V$, so by the induction hypothesis,

$$X(c_0, \kappa_0) \neq \bot.$$

Let $s_0 = X(c_0, \kappa_0)$. Since $F^\sharp(X) \sqsubseteq X$ and $(c_0, \kappa_0) \to_K (c, \kappa)$,

$$\begin{aligned}
&[\![\mathsf{cmd}(c)]\!](s_0) \\
&\sqsubseteq [\![\mathsf{cmd}(c)]\!](\bigsqcup\{X(c_1, \kappa_1) \mid (c_1, \kappa_1) \to_K (c, \kappa)\}) \\
&= F^\sharp(X)(c, \kappa) \\
&\sqsubseteq X(c, \kappa).
\end{aligned}$$

But $[\![\mathsf{cmd}(c)]\!](s') = \bot$ holds only if $s' = \bot$. Thus, $X(c, \kappa) \neq \bot$, as desired.

Next, using what we have just proved (i.e., (27)), we prove (26). Pick $(c, \kappa) \in \mathbb{C}_K$. If $(c, \kappa) \notin V$, then $\mathcal{X}(c, \kappa) = \bot$, so the desired inequality above follows immediately. Otherwise,

$$\mathcal{X}(c, \kappa) \neq \bot \ \wedge \ X(c, \kappa) \neq \bot,$$

where the first disequality comes from the definition of $\mathcal{X}$ and the second from (27). Now pick $x \in \mathsf{Var}$. Our proof obligation is now reduced to showing

$$\mathcal{X}(c, \kappa)(x) \sqsubseteq X(c, \kappa)(x).$$

This inequality is immediate if $\mathcal{X}(c, \kappa)(x) = \bot_v$. Suppose that

$$\mathcal{X}(c, \kappa)(x) \neq \bot_v.$$

Let $v = \mathcal{X}(c, \kappa)(x)$. Since $(c, \kappa) \in V$ and the domain of abstract values $\mathbb{V}$ is totally ordered, there exist

$$((c_0, \kappa_0), x_0), \ldots, ((c_n, \kappa_n), x_n)$$

such that

$$\begin{array}{l}((c_0, \kappa_0), x_0) \in \Omega_v \ \wedge\ (c_0, \kappa_0) \in V \\ \ \wedge\ (\forall 0 \le i < n.\, ((c_i, \kappa_i), x_i) \hookrightarrow_K ((c_{i+1}, \kappa_{i+1}), x_{i+1})) \\ \ \wedge\ ((c_n, \kappa_n), x_n) = ((c, \kappa), x).\end{array} \tag{28}$$

Note that every (c_i, κ_i) is in V. So, by (27),

$$\forall 0 \le i \le n.\, X(c_i, \kappa_i) \neq \bot. \tag{29}$$

We will show that

$$\begin{array}{l}v \sqsubseteq X(c_0, \kappa_0)(x_0) \\ \ \wedge\ (\forall 0 \le i < n.\ X(c_i, \kappa_i)(x_i) \sqsubseteq X(c_{i+1}, \kappa_{i+1})(x_{i+1})).\end{array} \tag{30}$$

Note that this gives the desired relationship $v \sqsubseteq X(c, \kappa)(x)$ because of the transitivity of $\sqsubseteq$.

The key to show the first conjunct in (30) is to notice that

$$\begin{array}{l}((c_0, \kappa_0) = (\iota, \epsilon) \ \wedge\ v = \top_v) \quad \vee \\ (\exists e.\ \mathsf{cmd}(c_0) = (x_0 := e) \ \wedge\ \mathsf{const}(e) = v).\end{array}$$

If the first disjunct holds, we can use our assumption that

$$s_I^\sharp \sqsubseteq X(\iota, \epsilon)$$

and derive that

$$v = \top_v = s_I^\sharp(x_0) \sqsubseteq X(c_0, \kappa_0)(x_0).$$

Assume that the disjunct holds. Since $X(c_0, \kappa_0) \neq \bot$, $(c_0, \kappa_0) = (\iota, \epsilon)$ or there exists some (c_0', κ_0') such that

$$(c_0', \kappa_0') \rightarrow_K (c_0, \kappa_0) \ \wedge\ X(c_0', \kappa_0') \neq \bot.$$

Since $s_I^\sharp \sqsubseteq X(\iota, \epsilon)$ and $F^\sharp(X) \sqsubseteq X$, in both cases, we have that

$$v \sqsubseteq X(c_0, \kappa_0)(x_0).$$

We now move on to the second conjunct of (30). In this case, we use a general fact that if

$$((c', \kappa'), x') \hookrightarrow_K ((c'', \kappa''), x'') \ \wedge\ X(c', \kappa') \neq \bot, \tag{31}$$

then

$$F^\sharp(X)(c'', \kappa'') \neq \bot \ \wedge\ X(c', \kappa')(x') \sqsubseteq F^\sharp(X)(c'', \kappa'')(x'').$$

Since $F^\sharp(X) \sqsubseteq X$, the second conjunct above implies that

$$X(c', \kappa')(x') \sqsubseteq X(c'', \kappa'')(x'').$$

Hence, the second conjunct of (30) follows if we discharge the condition (31) for consecutive elements in the sequence

$$((c_0, \kappa_0), x_0), \ldots, ((c_n, \kappa_n), x_n).$$

This condition holds because of (28) and (29). □

LEMMA A.9. *For every $c \in \mathbb{C}$,*

$$\mathsf{PA}_K(c) = \bigsqcup_{\kappa \in \mathbb{C}_c^*} \mathcal{X}(c, \kappa).$$

PROOF. Pick $c \in \mathbb{C}$. Recall the definition of the set of reachable nodes C in (8):

$$C = \{c \mid \exists \kappa.\, (\iota, \epsilon) \to_K^* (c, \kappa)\}.$$

If $c \notin C$, then

$$\forall \kappa \in \mathbb{C}_c^*.\, (c, \kappa) \notin V.$$

Hence, in this case,

$$\bigsqcup_{\kappa \in \mathbb{C}_c^*} \mathcal{X}(c, \kappa) = \bot.$$

But $\mathsf{PA}_K(c)$ is also $\bot$ by the definition of PA_K.

Suppose that

$$c \in C.$$

Let $K_0 = \{\kappa \mid (c, \kappa) \in V\}$. Pick $x \in \mathsf{Var}$. We will show

$$\mathsf{PA}_K(c)(x) = \bigsqcup_{\kappa \in K_0} \mathcal{X}(c, \kappa)(x). \tag{32}$$

The left hand side of this equation is the join of the set

$$V_L = \{v \in \mathbb{V} \mid \exists (c_0, x_0) \in \Theta_v.\, (c_0, x_0) \hookrightarrow_K^\dagger (c, x)\}. \tag{33}$$

The right hand side of the equation in (32) is the join of the set

$$\begin{aligned} V_R = \{v \in \mathbb{V} \mid \exists \kappa \in K_0.\, \exists ((c_0, \kappa_0), x_0) \in \Omega_v.\, & \\ (c_0, \kappa_0) \in V \ \wedge & \\ ((c_0, \kappa_0), x_0) \hookrightarrow_K^* ((c, \kappa), x)\}. & \end{aligned} \tag{34}$$

It suffices to prove that $V_L = V_R$. By the definitions of Ω_v and Θ_v,

$$(c_0, x_0) \in \Theta_v \iff ((c_0, \kappa_0), x_0) \in \Omega_v.$$

Hence,

$$\begin{aligned} V_R = \{v \in \mathbb{V} \mid \exists (c_0, x_0) \in \Theta_v.\, \exists \kappa_0.\, \exists \kappa \in K_0.\, & \\ (c_0, \kappa_0) \in V \ \wedge & \\ ((c_0, \kappa_0), x_0) \hookrightarrow_K^* ((c, \kappa), x)\}. & \end{aligned}$$

Also, by the definitions of V, K_0 and $(\hookrightarrow_K^\dagger)$,

$$(c_0, x_0) \hookrightarrow_K^\dagger (c, x)$$

if and only if

$$\exists \kappa_0.\, \exists \kappa \in K_0.\, (c_0, \kappa_0) \in V \ \wedge \ ((c_0, \kappa_0), x_0) \hookrightarrow_K^* ((c, \kappa), x).$$

Thus,

$$V_R = \{v \in \mathbb{V} \mid \exists (c_0, x_0) \in \Theta_v.\, (c_0, x_0) \hookrightarrow_K^\dagger (c, x)\}.$$

We have just shown that $V_R = V_L$, as desired. □