Selective Context-Sensitivity Guided by Impact Pre-Analysis

Hakjoo Oh[1] Wonchan Lee[1] Kihong Heo[1] Hongseok Yang[2] Kwangkeun Yi[1]
Seoul National University[1], University of Oxford[2]

Abstract

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.

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

Keywords Static analysis, context-sensitive analysis

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 [16, 17] 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 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. Our method can be instantiated with a range of static analyses, and provides a guideline for designing impact pre-analyses for them, in particular, an efficient way of implementing those pre-analyses based on graph reachability.

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 from 6.6 to 48.3%, with average 24.4%, compared with the baseline context-insensitive analysis, while increasing the analysis cost from 9.4 to 50.5%, with average 27.8%.

The general principle behind the design and the use of our impact pre-analysis can be used for developing other types of selective analyses. We show its applicability by following the same principle and developing 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. We implemented this technique for the octagon analysis [11] and our experiments show that our selective octagon analysis achieves competitive cost-precision tradeoffs when applied to real-world benchmark programs.

Contributions

- We present a method for performing selective context-sensitive analysis that receives guidance from an impact pre-analysis.
- We show that the general idea behind our selective method is not limited to context-sensitivity. We present a selective relational analysis that is guided by an impact pre-analysis.
- We experimentally show the effectiveness of selective analyses designed according to our method, with real-world C programs.

PLDI'14, June 9–11, 2014, Edinburgh, United Kingdom

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`. Procedure `xmalloc` is a wrapper of the `malloc` function. It is called twice in procedure `multi_glob`, once with the argument `size` (line 4) and again with an input from the environment (line 6). The `main` routine of this program calls procedure `f` and `g`. Procedure `multi_glob` is called in `f` and `g` with different argument values.

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 asks a similar question, but this time for the pointer variable `q`. Note that the first query always holds, but the second query is not necessarily true.

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, and the analysis concludes that `xmalloc` allocates a buffer of size in $[-\infty, +\infty]$.

Context-sensitive analysis A natural way to fix this precision issue is to increase the context-sensitivity. One popular approach is k-CFA analysis [16, 17]. 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, 3-CFA analyzes the procedure `xmalloc` separately for each of the following calling contexts:

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

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. Note that the 3-CFA analysis can prove the first query: the analysis analyzes the first four contexts separately and infers that a buffer of size greater than 1 gets allocated under these calling contexts.

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, for every $k \geq 0$, the k-CFA analysis concludes that `p` points to a buffer of size $[-\infty, +\infty]$. Also, it is unnecessary to analyze `g` separately for call sites 16 and 17, because those two calls have the same effect on the query.

Our selective context-sensitivity With our approach, an analysis can analyze 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`:

$$4\cdot 10\cdot 14, \quad 4\cdot 10\cdot 15, \quad 4\cdot 11, \quad \text{all the other contexts} \tag{2}$$

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

```
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 }
```

Figure 1. Example Program

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 is able to prove the first query.

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 abstract domain and transfer functions, and can be run efficiently even with full context-sensitivity.

For instance, we approximate the interval analysis in this example using a pre-analysis with two abstract values: $\bigstar$ and $\top$. 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:

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

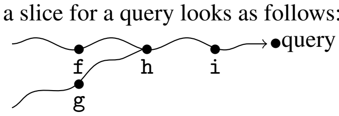
This simple abstract domain of the pre-analysis is chosen because we are interested in showing the absence of buffer overruns and the analysis proves such properties only when it finds non-negative intervals for buffer sizes and indices.

We run this pre-analysis under full context-sensitivity (i.e., ∞-CFA). For our example program, we obtain a summary of the procedure `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 ★ 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 Next, 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 ★ in the pre-analysis run. In our example, the pre-analysis assigns ★ to the 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 procedure `f`, `g`, `h`, and `i`, respectively. However, if the slice involves a recursive call, 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, and that all the calling contexts of `xmalloc` in this slice are: $4\cdot 10\cdot 14$, $4\cdot 10\cdot 15$, and $4\cdot 11$. Our analysis decides to distinguish these contexts and their suffixes.

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 a nonnegative interval to the expression `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, although the call to `xmalloc` at line 6 is analyzed in a context-insensitive way, our analysis ensures that `xmalloc` in this case returns only to line 6, not to line 4.

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

For instance, suppose that we want to develop a selective version of the octagon analysis, which tracks only some relationships between program variables that are likely to be tracked well by the octagon analysis and also to help the proofs of given queries. 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. In Section 6, we describe this selective octagon analysis in detail.

3. Program Representation

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:

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

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 assume for simplicity that there are no indirect function calls such as calls via function pointers.

We associate a primitive command with each node c of our control flow graph, and denote it by $\mathsf{cmd}(c)$. For brevity, we consider simple primitive commands specified by the following grammar:

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

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.

For simplicity, we handle parameter passing and return values of procedures via simple syntactic encoding. 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 Context-Sensitivity Parameter K

We consider selective context-sensitive analyses specified by the following data: (1) a domain $\mathbb{S}$ of abstract states, which forms a complete lattice structure $(\mathbb{S}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$; (2) an initial abstract state $s_I \in \mathbb{S}$ at the entry of the `main` procedure; (3) a monotone abstract semantics of primitive commands $[\![cmd]\!] : \mathbb{S} \rightarrow \mathbb{S}$; (4) a context selector K that maps procedures to sets of calling contexts (sequences of call nodes):

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

For each 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 $\kappa \in K$ for $\kappa \in \bigcup_{f\in\mathbb{F}} K(f)$.

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 $(\rightarrow) \subseteq \mathbb{C}\times\mathbb{C}$ is extended to $\rightarrow_K$ on $\mathbb{C}_K$:

Definition 1 ($\rightarrow_K$)**.** *$(\rightarrow_K) \subseteq \mathbb{C}_K \times \mathbb{C}_K$ is the context-enriched control flow relation:*

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

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

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

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 1. *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* $K = \{\texttt{main} \mapsto \{\epsilon\}, \texttt{f} \mapsto \{14, 15\}, \texttt{g} \mapsto \{\epsilon\}, \texttt{multi\_glob} \mapsto \{10 \cdot 14, 10 \cdot 15, 11\}, \texttt{xmalloc} \mapsto \{4 \cdot 10 \cdot 14, 4 \cdot 10 \cdot 15, 4 \cdot 11, \epsilon\}\}$.

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

$$\mathbb{D} = (\mathbb{C}_K \to \mathbb{S}) \tag{3}$$

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:

$$F(X)(c, \kappa) = [\![\mathsf{cmd}(c)]\!]\Big(\bigsqcup_{(c_0,\kappa_0) \to_K (c,\kappa)} X(c_0, \kappa_0)\Big). \tag{4}$$

The static analysis computes an abstract element $X \in \mathbb{D}$ satisfying the following condition:

$$s_I \sqsubseteq X(\iota, \epsilon) \ \wedge\ \forall (c, \kappa) \in \mathbb{C}_K.\ F(X)(c, \kappa) \sqsubseteq X(c, \kappa) \tag{5}$$

In general, many X can satisfy the condition in (5). Some analyses compute the least X satisfying (5). Other analyses use a widening operator [1], $\bigtriangledown : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, and compute not necessarily the least, but some solution of (5).

Example 2 (Interval Analysis)**.** *The interval analysis is a standard example that uses a widening operator. Let* $\mathbb{I}$ *be the domain of intervals:* $\mathbb{I} = \{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, +\infty\} \ \wedge\ l \le u\}$. *Using this domain, we specify the rest of the analysis:*

1. *The abstract states are* $\bot$ *or functions from program variables to their interval values:* $\mathbb{S} = \{\bot\} \cup (\mathsf{Var} \to \mathbb{I})$
2. *The initial abstract state is:* $s_I(x) = [-\infty, +\infty]$.
3. *The abstract semantics of primitive commands is:*

$$[\![skip]\!](s) = s, \quad [\![x := e]\!](s) = \begin{cases} s[x \mapsto [\![e]\!](s)] & (s \neq \bot) \\ \bot & (s = \bot) \end{cases}$$

where $[\![e]\!]$ *is the abstract evaluation of the expression* e*:*

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

4. *The last component of the analysis is a widening operator, which is defined as a pointwise lifting of the following widening operators* $\bigtriangledown_I : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ *for intervals:*

$$\begin{array}{rcl} [l, u] \bigtriangledown_I [l', u'] & = & [\mathrm{ite}(l' < l, \mathrm{ite}(l' < 0, -\infty, 0), l), \\ & & \ \mathrm{ite}(u' > u, +\infty, u)] \end{array}$$

where $\mathrm{ite}(p, a, b)$ *evaluates to* a *if* p *is true and* b *otherwise. The above widening operator uses* 0 *as a threshold, which is useful when proving the absence of buffer overruns.*

Queries Queries are triples in $\mathcal{Q} \subseteq \mathbb{C} \times \mathbb{S} \times \mathsf{Var}$, and they are given as 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 the variable x. For instance, in the interval analysis, a typical query is

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

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:

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

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 partial context-sensitivity, using the induced abstract domain $\mathbb{D}$ and transfer function $F : \mathbb{D} \to \mathbb{D}$ for this program in (3) and (4). We assume that the analysis employs the fixpoint algorithm based on widening operation $\bigtriangledown : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$.

How should we automatically choose an effective 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 to use the results of this pre-analysis for constructing an effective context selector 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:

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

It computes the least X satisfying

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

What is less standard is the soundness and efficiency conditions for our pre-analysis, which provides a guideline on the design of these pre-analyses. 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.[1] This is identical to the standard soundness requirement of a static program analysis,

[1] 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 and the resulting selective main analysis is hardly cost-effective.

except that the condition is stated not over the concrete semantics of a given program, but over the main analysis. The condition has the following four requirements:

1. There should be a concretization function $\gamma : \mathbb{S}^\sharp \rightarrow \wp(\mathbb{S})$. 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 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: 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 1. *Let $M \in \mathbb{D}$ be the main analysis result, i.e., a solution of* (5) *under full context-sensitivity ($K = K_\infty$). Let $P \in \mathbb{D}^\sharp$ be the pre-analysis result, i.e., the least solution of* (6)*. Then, $\forall c \in \mathbb{C}, \kappa \in \mathbb{C}_c^*.\ M(c, \kappa) \in \gamma(P(c, \kappa))$.*

Efficiency condition The next condition is for the efficiency of our pre-analysis. It consists of two requirements, and ensures that the pre-analysis can be computed using efficient algorithms:

1. The abstract states are $\bot$ or functions from program variables to abstract values: $\mathbb{S}^\sharp = \{\bot\} \cup (\mathsf{Var} \rightarrow \mathbb{V})$, where $\mathbb{V}$ is a finite complete lattice $(\mathbb{V}, \sqsubseteq_v, \bot_v, \top_v, \sqcup_v, \sqcap_v)$. An initial abstract state is $s_I^\sharp = \lambda x.\ \top_v$.
2. The abstract semantics of primitive commands has a simple form involving only join operation and constant abstract value, which is defined as follows:
$$[\![skip]\!]^\sharp(s) = s, \quad [\![x := e]\!]^\sharp(s) = \begin{cases} s[x \mapsto [\![e]\!]^\sharp(s)] & (s \neq \bot) \\ \bot & (s = \bot) \end{cases}$$
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
$$\mathsf{var}(e) = \{x_1, \ldots, x_n\}, \qquad \mathsf{const}(e) = v.$$

Example 3 (Impact Pre-Analysis for the Interval Analysis)**.** *We design a pre-analysis for our interval analysis in Example 2, which satisfies our soundness and efficiency 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 the function $\gamma_v : \{\bot_v, \bigstar, \top_v\} \rightarrow \wp(\mathbb{I})$ as follows:*
$$\gamma_v(\top_v) = \mathbb{I}, \quad \gamma_v(\bigstar) = \{[a, b] \in \mathbb{I} \mid 0 \leq a\}, \quad \gamma_v(\bot_v) = \emptyset$$
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 intervals according to this function. We include such a case because non-negative intervals, not negative ones, prove buffer-overrun properties.
2. *The domain of abstract states is defined as $\mathbb{S}^\sharp = \{\bot\} \cup (\mathsf{Var} \rightarrow \mathbb{V})$. The meaning of abstract states in $\mathbb{S}^\sharp$ is given by γ such that $\gamma(\bot) = \{\bot\}$ and, for $s^\sharp \neq \bot$,*
$$\gamma(s^\sharp) = \{s \in \mathbb{S} \mid s = \bot \ \vee \ \forall x \in \mathsf{Var}.\ s(x) \in \gamma_v(s^\sharp(x))\}.$$
3. *Initial abstract state: $s_I^\sharp = \top = \lambda x. \top_v$.*
4. *Abstract evaluation $[\![e]\!]^\sharp$ of expression e: for every $s \neq \bot$,*
$$[\![n]\!](s) = \mathrm{ite}(n \geq 0, \bigstar, \top_v), \quad [\![e_1 + e_2]\!](s) = [\![e_1]\!](s) \sqcup_v [\![e_2]\!](s)$$
$$[\![x]\!](s) = s(x), \quad [\![e_1 - e_2]\!](s) = \top_v$$
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 $\top_v$.

Running the pre-analysis via reachability-based algorithm The class of our pre-analyses enjoys efficient algorithms (e.g., [3, 15]) for computing the least solution X that satisfies (6), even though it is fully context-sensitive. For our purpose, we provide a variant of the graph reachability-based algorithm in [15]. Our algorithm is specialized for our pre-analysis and is more efficient than the algorithm in [15]. 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:
$$\Theta = \mathbb{C} \times \mathsf{Var}, \qquad (\hookrightarrow) \subseteq \Theta \times \Theta$$
The node set consists of pairs of program nodes and variables, and $(\hookrightarrow)$ is the edge relation between the nodes.

Definition 2 $(\hookrightarrow)$**.** *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 values of variables flow to other variables in each primitive command:*
$$(c, x) \hookrightarrow (c', x') \ \textit{iff}$$
$$\begin{cases} c \rightarrow c' \ \wedge \ x = x' & (\mathsf{cmd}(c') = skip) \\ c \rightarrow c' \ \wedge \ x = x' & (\mathsf{cmd}(c') = y := e \ \wedge \ y \neq x') \\ c \rightarrow c' \ \wedge \ x \in \mathsf{var}(e) & (\mathsf{cmd}(c') = y := e \ \wedge \ y = x') \end{cases}$$

We can extend the $\hookrightarrow$ to its context-enriched version $\hookrightarrow_K$:

Definition 3 $(\hookrightarrow_K)$**.** *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:*
$$((c, \kappa), x) \hookrightarrow_K ((c', \kappa'), x') \ \textit{iff}$$
$$\begin{cases} (c, \kappa) \rightarrow_K (c', \kappa') \ \wedge \ x = x' & (\mathsf{cmd}(c') = skip) \\ (c, \kappa) \rightarrow_K (c', \kappa') \ \wedge \ x = x' & (y \neq x') \\ (c, \kappa) \rightarrow_K (c', \kappa') \ \wedge \ x \in \mathsf{var}(e) & (y = x') \end{cases}$$
$$\textit{(where } \mathsf{cmd}(c') \textit{ in the last two cases is } y := e)$$

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

Definition 4 $(\hookrightarrow_K^\dagger)$**.** *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:*
$$(c, x) \hookrightarrow_K^\dagger (c', x') \ \textit{iff}$$
$$\exists \kappa, \kappa'.\ (\iota, \epsilon) \rightarrow_K^* (c, \kappa) \ \wedge \ ((c, \kappa), x) \hookrightarrow_K^* ((c', \kappa'), x').$$

We use the tabulation algorithm in [15] for computing $(\hookrightarrow^{\dagger}_{K})$. While computing $(\hookrightarrow^{\dagger}_{K})$, the algorithm also collects the set C of reachable nodes:

$$C = \{c \mid \exists\kappa.\, (\iota, \epsilon) \rightarrow^{*}_{K} (c, \kappa)\}. \tag{7}$$

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:

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

Finally, using $(\hookrightarrow^{\dagger}_{K})$ and Θ_v, the algorithm constructs PA_K:

Definition 5 (PA_K)**.** $\mathsf{PA}_K \in \mathbb{C} \rightarrow \mathbb{S}^{\sharp}$ *is defined as follows:*

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

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

Lemma 2. *Let X be the least solution satisfying* (6)*. Then,* $\mathsf{PA}_K(c) = \bigsqcup_{\kappa \in \mathbb{C}^*} X(c, \kappa)$.

Our reachability-based algorithm is $|\mathbb{V}|^3$-times faster in the worst case than the RHS algorithm [15]. The algorithm in [15] works on a graph with the following set of vertices:

$$\Theta' = \{(c, s) \mid c \in \mathbb{C} \wedge s \neq \bot \wedge (\exists x. \forall y. y \neq x \implies s(y) = \bot_v)\}$$

Note that the set Θ' is $|\mathbb{V}|$-times larger than set Θ used in our algorithm and the worst-case time complexity is cubic on the size of the underlying graph [15].

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 10), 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 1).

Query selection We first select queries that can benefit from increased context-sensitivity. Among given queries $\mathcal{Q} \subseteq \mathbb{C} \times \mathbb{S} \times \mathsf{Var}$ about the given program, we select the following ones:

$$\begin{aligned}\mathcal{Q}^{\sharp} = \{&(c, x) \in (\mathbb{C} \times \mathsf{Var}) \mid \exists s \in \mathbb{S}. \\ &(c, s, x) \in \mathcal{Q} \ \wedge\ \forall s' \in \gamma(\mathsf{PA}_{K_\infty}(c)).\ s \sqcup s' \neq \top\}\end{aligned} \tag{8}$$

where $\mathsf{PA}_{K_\infty} : \mathbb{C} \rightarrow \mathbb{S}^{\sharp}$ is the pre-analysis result. 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 (8) 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 $\mathsf{K} : \mathbb{F} \rightarrow \wp(\mathbb{C}^*_c)$. 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 10) that answers this question (Proposition 1).

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 6 (Φ)**.** *Sources Φ are vertices in Θ where dependencies begin:* $\Phi = \{(c_0, x_0) \in \Theta \mid \neg(\exists(c, x) \in \Theta.\ (c, x) \hookrightarrow (c_0, x_0))\}$.

We compute the set $\Phi_{(c_q, x_q)}$ of sources on which the query (c_q, x_q) depends:

Definition 7 ($\Phi_{(c_q, x_q)}$)**.** *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 4. *Consider the control flow graph in Figure 2. Node* 6 *denotes the query point, i.e.,* $(c_q, x_q) = (6, \mathtt{z})$*. The gray nodes represent the sources on which the query depends, i.e.,* $\Phi_{(6,\mathtt{z})} = \{(1, \mathtt{x}), (7, \mathtt{y})\}$.

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

$$((c_0, \kappa_0), x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q, \kappa_q), x_q) \tag{9}$$

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

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

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

Example 5. *In Figure 2, suppose that κ_0 and κ_1 are the initial contexts of procedures* $\mathtt{m}$ *and* $\mathtt{h}$*, respectively. For source* $(1, \mathtt{x})$*, we find the following dependency path to the query* $(6, \mathtt{z})$*:*

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

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

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

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

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

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

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. 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

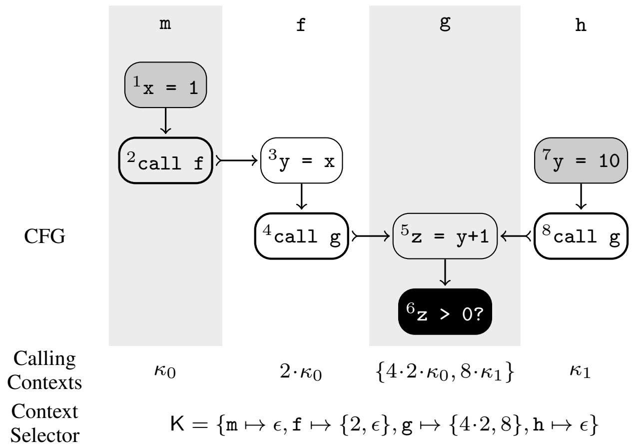


Figure 2. Example context selector. Gray and black nodes in CFG are source and query points, respectively.

suffix of each other, for instance $\kappa_0 = c_2 \cdot c_1$ and $\kappa_i = c_3 \cdot c_1$. In this case, $\kappa_i \ominus \kappa_0 = c_3$.

Assumption 1. *In general, the above definition of partial contexts requires that the input program should be well-formed with respect to the query: for a path* (10) *from a source to the query, every call site, $c_i \in \mathbb{C}_c$, in that path should not be included in the initial context κ_0. We require this condition because our selective context-sensitive analysis aims at distinguishing only the calls after passing the sources of dependency and analyzing context-insensitively those encountered before reaching those sources, which do not contribute to the query. This well-formedness assumption is not a strong restriction and its violation nearly never happens in practice. We did not observe any violation of the assumption in our benchmark programs (Section 7). If the program is not well-formed to a query, then we simply ignore it.*

Let us explain the condition with an example. Suppose that $\kappa_0 = c_3 \cdot c_2 \cdot c_1$ is the initial context at c_0 and $\kappa_i = c_1$ is the context at c_i. Suppose further that c_i is a call node. Then, our condition requires that c_i should not be one of call site c_1, c_2, and c_3. Formally, the condition is defined as follows:

We say the given program is well-formed with respect to the query (c_q, x_q) iff for every $(c_0, x_0) \in \Phi_{(c_q,x_q)}$ and its valid value-flow path

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

for all $0 \le i \le n$ such that $c_i \in \mathbb{C}_c$, c_i is not included in the initial calling context κ_0; i.e.,

$$c_i \notin \kappa_0 \tag{11}$$

where we write $c \in \kappa$ when there exists some κ' such that $c \cdot \kappa'$ is a suffix of κ.

In summary, for the path in (10), 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 (10) as follows:

Definition 9 (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 $(\iota, \epsilon) \to^*_{K_\infty} (c_0, x_0)$. 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 6. *From the path p_1 in Example 5, 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:*

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

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

Definition 10 (K, Context Selector)**.** *Let (c_q, x_q) be a query. The context selector $\mathsf{K} \in \mathbb{F} \to \wp(\mathbb{C}_c^*)$ 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, $\mathcal{E}(f) = \emptyset$.

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 1 (Impact Realization)**.** *Let $\mathsf{PA}_{K_\infty} \in \mathbb{C} \to \mathbb{S}^\sharp$ be the result of the impact pre-analysis (Definition 5). Let $q \in \mathcal{Q}^\sharp$ be a selected query* (8)*. Let K be the context selector for q (Definition 10) 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 $(q \overset{let}{=} (c, x))$

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

This impact realization holds thanks to two key properties. First, our selective context-sensitivity K (Definition 10) 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 Selective Relational Analysis

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 use the same principle to develop a selective relational analysis with the octagon domain [11].

Overview Consider the following code snippet:

```
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 }
```

The first query at line 4 always holds but the second one at line 5 is not necessarily true.

A fully relational octagon analysis, which tracks contraints of the form $\pm x \pm y \le c$ (where $c \in \mathbb{Z} \cup \{\infty\}$) between *all* variables

x and y, can prove the first query. The analysis infers constraints $\mathtt{b} - \mathtt{a} \leq 0$ at line 1 and $\mathtt{i} - \mathtt{b} \leq -1$ at line 3. Then, combining the two via a closure operation [11], the analysis concludes that constraint $\mathtt{i} - \mathtt{a} \leq -1$ holds at line 4. More specifically, the fully relational octagon analysis computes the table (i.e., difference bound matrix [11]) on the left side of the following:

	a	b	c	i
a	0	0	∞	-1
b	0	0	∞	-1
c	∞	∞	0	∞
i	∞	∞	∞	0

	a	b	c	i
a	★	★	$\top$	★
b	★	★	$\top$	★
c	$\top$	$\top$	★	$\top$
i	$\top$	$\top$	$\top$	★

(13)

where the bound c in constraint $x - y \leq c$ is stored at row y and column x in the table.[2] Note that the (a,i) entry of the table stores -1, which means that the analysis proves $\mathtt{i} - \mathtt{a} \leq -1$ 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 $x + y \sqsubseteq \bigstar$ 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 the table on the righthand side in (13).

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 $\mathtt{i} - \mathtt{a} \sqsubseteq \bigstar$. Our dependency analysis figures out that the constraint was derived in the pre-analysis by combining two constraints $\mathtt{i}-\mathtt{b} \sqsubseteq \bigstar$ and $\mathtt{b}-\mathtt{a} \sqsubseteq \bigstar$ in its closure operation. Therefore, the dependency analysis concludes that the main analysis should be able to derive three relationships $\mathtt{i} - \mathtt{a} \sqsubseteq \bigstar$, $\mathtt{i} - \mathtt{b} \sqsubseteq \bigstar$, and $\mathtt{b} - \mathtt{a} \sqsubseteq \bigstar$ 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 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 [11] 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 [11].

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 $\gamma_{\mathbb{V}}$ such that $\gamma_{\mathbb{V}}(\bigstar) = \mathbb{Z}$ and $\gamma_{\mathbb{V}}(\top) = \mathbb{Z} \cup \{+\infty\}$. 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.,

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

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

[2] 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 $(2 \times |\mathsf{Var}|)^2$.

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.,

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

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 $\left((\mathsf{lfp}F^\natural)(c)\right)_{xy} = \left(\bigstar, S\right)$. Finally, we extract the packing configuration Π using $\pi_{(c,x,y)}$ as follows:

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

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

Proposition 2 (Impact Realization)**.** *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, $\left((\mathsf{lfp}F^\Pi)(c)(\pi_{(c,x,y)})\right)_{xy} \neq +\infty$.*

7. Experiments

Selective Context-Sensitive Analysis In experiments, we used SPARROW [19], a buffer-overrun analyzer that supports the 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.

On top of the baseline analyzer, we have implemented our technique: we implemented the impact pre-analysis in Example 3 and extended the baseline analysis to be selectively context-sensitive. 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.

We have run the analysis for 10 software packages from the GNU open-source projects. The analysis is global: the entire program is analyzed starting from the `main` procedure. All experiments were done on a Linux 2.6 system running on a single core of Intel 3.07GHz box with 24GB of memory.

Table 1 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.

The results show that our method leads to a cost-effective improvement of the analysis precision. In total, the context-insensitive interval analysis points out 12,701 buffer accesses as potential buffer-overrun errors (there is a total of 83,776 buffer accesses in the 10 programs). 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 not uncommon in the interval analysis of C programs. Our pre-analysis is able to prescribe such a long call sequence as context-sensitive targets. For instance, in `a2ps-4.14`, among 1682 call sequences prescribed by our pre-analysis, 488 call sequences were of length longer than or equal to 3.

According to our experience, the k-callstrings approach does 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`.

Selective Octagon Analysis We have implemented our selective method 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 [11, 13]. The syntactic packing approach relates variables together if they are involved in the same syntactic block [11]. We limited the maximum pack size by 10 in the syntactic packing strategy, since otherwise the analysis did not scale.

Table 2 shows our benchmark programs. Note that, although a relational analysis is a key to proving important numerical properties, it is useful only for specific target programs and queries [4, 11]. Thus, we first identified a set of benchmark programs and their buffer-overrun queries whose proofs require relational information, and compared the performance of the two analyses on these programs and queries. Column **#Query** shows the number of relational queries that we consider in our experiments. In the experiments, we manually in-lined 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 135 queries in total, our analysis is able to prove 132 (97.8%) queries in 3,632.7s. 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.

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 memory costs for representing $2|\mathsf{Var}| \times 2|\mathsf{Var}|$ matrices and the expensive strong closure operation [11] 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 [11] instead of the strong closure. In our experiments, using the shortest-path closure made no difference in the pre-analysis precision.

8. Related Work

Most of the previous context-sensitive analysis techniques assign contexts to calls in a uniform manner. The k-callstring approach (or k-CFA) [16, 17] and its flexible variants [6], k-object sensitivity [10], and type sensitivity [18] 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 [8], which employs multiple policies of assigning contexts in a single analysis, still does not tailor those policies to the program to analyze. There are also other approaches to context-sensitivity based on function summaries like [15], but here we do not discuss them as it is by itself a challenge to design a summary-based analysis with abstract domains of infinite height.

While refinement-based analyses [5, 14, 20] 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 [5]) 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

Program	LOC	Proc	Context-Insensitive		Our Selective Context-Sensitive Analysis						Alarm	Overhead	
			#alarm	time	#alarm	pre	main	total	#selected call-sites	$\rightsquigarrow$	reduction	pre	main
spell-1.0	2,213	31	58	0.6	30	0.1	0.8	0.9	25 / 124 (20.2 %)	3	48.3%	16.7%	33.3%
bc-1.06	13,093	134	606	14.0	483	1.9	14.3	16.2	29 / 777 (3.7 %)	2	20.3%	13.6%	2.1%
tar-1.17	20,258	222	940	42.1	799	5.4	41.8	47.2	51 / 1213 (4.2 %)	3	15.0%	12.8%	−0.7%
less-382	23,822	382	654	123.0	562	3.3	163.1	166.4	51 / 1,522 (3.4 %)	4	14.1%	2.7%	32.6%
sed-4.0.8	26,807	294	1,325	107.5	1,238	7.4	110.2	117.6	25 / 868 (2.9 %)	3	6.6%	6.9%	2.5%
make-3.76	27,304	191	1,500	84.4	1,028	7.1	99.1	106.2	67 / 1,050 (6.4 %)	3	31.5%	8.4%	17.4%
grep-2.5	31,495	153	735	12.1	653	2.4	13.5	15.9	33 / 530 (6.2 %)	3	11.2%	19.8%	11.6%
wget-1.9	35,018	434	1,307	69.0	942	12.5	69.6	82.1	79 / 1,973 (4.0 %)	5	27.9%	18.1%	0.9%
a2ps-4.14	64,590	980	3,682	118.1	2,121	29.5	148.2	177.7	237 / 2,450 (9.7 %)	9	42.4%	25.0%	25.5%
bison-2.5	101,807	1,427	1,894	136.3	1,742	34.6	138.8	173.4	173 / 2,038 (8.5 %)	4	8.0%	25.4%	1.8%
Total	346,407	4,248	12,701	707.1	9,598	104.2	799.4	903.6	770 / 12,545 (6.1 %)		**24.4%**	**14.7%**	**13.1%**

Table 1. 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	LOC	#Variable	#Query	Syntactic Packing Approach				Our Selective Relational Analysis						Comparison	
				proven	time	mem	pack	proven	pre	main	total	mem	pack	Precision	Time
calculator-1.0	298	197	10	2	0.3	63	18 (7.3)	10	0.1	0.1	0.2	52	3 (3.6)	+8	-33.3%
spell-1.0	2,213	531	16	1	4.8	109	119 (7.7)	16	1.7	0.7	2.4	63	6 (11.0)	+15	-50.0%
barcode-0.96	4,460	2,002	37	16	11.8	221	276 (8.1)	37	12.2	18.3	30.5	100	12 (25.0)	+21	158.5%
httptunnel-3.3	6,174	1,908	28	16	26.0	220	454 (7.0)	26	10.8	4.5	15.3	105	8 (5.8)	+10	-41.2%
bc-1.06	13,093	2,194	10	2	247.1	945	606 (7.8)	9	82.3	35.0	117.3	212	4 (4.0)	+7	-52.5%
tar-1.17	20,258	5,332	17	7	1,043.2	1,311	1,259 (7.5)	17	598.5	63.3	661.8	384	7 (3.9)	+10	-36.6%
less-382	23,822	4,482	13	0	3,031.5	1,439	1,017 (6.3)	13	2,253.2	596.2	2,849.4	955	8 (6.3)	+13	-6.0%
a2ps-4.14	64,590	16,531	11	0	29,479.3	2,304	2,608 (7.8)	11	2,223.5	518.2	2,741.7	909	6 (6.7)	+11	-90.7%
Total	135,008	33,177	142	44	33,840.3	6,611		139	5,182.3	1,236.3	6,418.6	2,780		+95	-81.0%

Table 2. 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.

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 [5, 14, 20].

Our approach is orthogonal to demand-driven analyses [7, 20, 21]. 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.

In a high level, our approach suggests a novel technique for *analysis-parameter inference* [9, 12, 22]. There are many parameters to tune in static analysis, to improve either precision or scalability. The problem is how to find a set of minimal, or at least sufficient, parameters for the goal. Liang et al. [9] use machine learning to find a minimal context-sensitivity for given queries. Guided by the number of queries each analysis run has proven, the machine learning algorithm infers a minimal k value for each function. However, they study the minimal abstraction itself and provide no practical solutions for selective context-sensitivity. Zhang et al. [22] 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. [12] use a dynamic analysis to select an appropriate parameter for a given query, while we use a static pre-analysis for parameter selection.

Our selective octagon analysis is similar to the existing octagon analyses (such as [4, 11, 13]) 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 blindly construct variable packs based on syntactic heuristics [11, 13] or program dependencies [4].

9. Conclusion

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 two 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. Our experiments with realistic benchmarks showed that the method reduces the number of false alarms of a 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, and our experiments showed that our selective octagon analysis proves 88 more queries than the existing one based on the syntactic variable packing and reduces the analysis cost by 81%. We believe that our approach can be used for developing other selective analyses as well, e.g., selective flow-sensitive analysis, selective loop-unrolling, etc.

Acknowledgements This work was partially 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 Samsung Electronics. Yang was supported by EPSRC.

References

[1] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 1992.

[3] Alain Deutsch. On the complexity of escape analysis. In *POPL*, 1997.

[4] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *POPL*, 2012.

[5] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *SAS*, 2003.

[6] Williams L. Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 1989.

[7] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *PLDI*, 2001.

[8] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI*, 2013.

[9] Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *POPL*, 2011.

[10] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, 2002.

[11] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[12] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. Abstractions from tests. In *POPL*, 2012.

[13] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI*, 2012.

[14] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA*, 1994.

[15] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[16] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[17] Olin Grigsby Shivers. *Control-flow analysis of higher-order languages -or- taming lambda*. PhD thesis, CMU, 1991.

[18] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, 2011.

[19] Sparrow. `http://ropas.snu.ac.kr/sparrow`.

[20] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, 2006.

[21] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, 2005.

[22] Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, 2013.

A. Proofs

In this appendix, we prove the impact guarantee (Proposition 1) of our selective context-sensitive analysis as well as the correctness of our reachability-based pre-analysis algorithm (Lemma 2).

Remark 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 2) 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* (6) *via our reachability algorithm.*

A.1 Proof of Proposition 1

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 3 and 6:

$$\begin{array}{cll} & \gamma(\top[x \mapsto \mathsf{PA}_{K_\infty}(c)(x)]) & \\ = & \gamma(\top[x \mapsto \mathsf{PA}_\mathsf{K}(c)(x)]) & \text{(Lemma 3)} \\ \supseteq & \gamma(\mathsf{PA}_\mathsf{K}(c)) & \text{(Soundness of } \top) \\ \ni & \mathsf{MA}_\mathsf{K}(\kappa, c) & \text{(Lemma 6)} \end{array}$$

□

What is non-trivial is the first equality (Lemma 3), 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 10), as far as the selected query (c, x) is concerned.

Lemma 3 (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 10). 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:

$$\begin{array}{l} \forall (c_0, x_0) \in \Phi. \\ \quad (c_0, x_0) \hookrightarrow^\dagger_{K_\infty} (c_q, x_q) \Longleftrightarrow (c_0, x_0) \hookrightarrow^\dagger_\mathsf{K} (c_q, x_q). \end{array}$$

- ($\Longrightarrow$) By Lemma 4.
- ($\Longleftarrow$) 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 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 4). Then, we prove that there is no $(c', x') \in \Phi \setminus \Phi_{(c_q, x_q)}$ such that $(c', x') \hookrightarrow^\dagger_\mathsf{K} (c_q, x_q)$ (Lemma 5). Comprising these two lemmas, we can prove Lemma 3.

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

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

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 $\kappa'_i = \kappa_i \ominus \kappa_0$.

Proof. By the definition of K,

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

We show that

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

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

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

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 Assumption 1,
   $$(c_i \cdot \kappa_i) \ominus \kappa_0 \neq \epsilon$$
   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
   $$\begin{aligned}\kappa_i' &= (\mathsf{callof}(c_{i+1}) \cdot \kappa_{i+1}) \ominus \kappa_0\\ &= \mathsf{callof}(c_{i+1}) \cdot (\kappa_{i+1} \ominus \kappa_0)\\ &= \mathsf{callof}(c_{i+1}) \cdot \kappa_{i+1}'\end{aligned}$$
   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
   $$\mathsf{callof}(c_{i+1}) \in \kappa_0 \tag{15}$$
   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 Assumption 1, $c_j = \mathsf{callof}(c_{i+1}) \notin \kappa_0$, which contradicts (15).

□

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 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

$$(c_0, \kappa_0) \rightarrow_{\mathsf{K}} \cdots \rightarrow_{\mathsf{K}} (c_q, \kappa_q). \tag{16}$$

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 4, we have a $\hookrightarrow_{\mathsf{K}}$-path

$$((c_s, \epsilon), x_s) \rightarrow_{\mathsf{K}} \cdots \hookrightarrow_{\mathsf{K}} ((c_q, \kappa_q), x_q) \tag{17}$$

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

$$(c_s, \epsilon) \rightarrow_{\mathsf{K}} \cdots \rightarrow_{\mathsf{K}} (c_q, \kappa_q). \tag{18}$$

This path should be distinct from the path (16). Let (c_i, κ_i) be the farthest point from the query that (16) and (18) agree. We further assume that we have chosen the path (17) such that among all the $\rightarrow_{\mathsf{K}}$-paths from (c_s, ϵ) to (c_q, κ_q), (18) has the longest common suffix with (16). Let (c_{i-1}, κ_{i-1}) and $(c_{i-1}', \kappa_{i-1}')$ be the first diverging point from the query such that

$$\begin{aligned}&(c_0, \kappa_0) \rightarrow_{\mathsf{K}} \cdots \rightarrow_{\mathsf{K}} (c_{i-1}, \kappa_{i-1})\\ &\qquad \rightarrow_{\mathsf{K}} (c_i, \kappa_i) \rightarrow_{\mathsf{K}} \cdots (c_q, \kappa_q)\end{aligned} \tag{19}$$

and

$$\begin{aligned}&(c_s, \epsilon) \rightarrow_{\mathsf{K}} \cdots \rightarrow_{\mathsf{K}} (c_{i-1}'', \kappa_{i-1}'')\\ &\qquad \rightarrow_{\mathsf{K}} (c_i, \kappa_i) \rightarrow_{\mathsf{K}} \cdots (c_q, \kappa_q)\end{aligned} \tag{20}$$

where $(c_{i-1}, \kappa_{i-1}) \neq (c_{i-1}'', \kappa_{i-1}'')$. Note that (16) and (18) 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

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

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

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

whenever we have path (20). Therefore, (c_i, κ_i) cannot be the farthest point.

2. When $c_i \in \mathbb{C}_e$:
By the definition of $\to_{\mathsf{K}}$,

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

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 (18) 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

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

From the Assumption 1, 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 $\to_{\mathsf{K}}$,

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

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 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* (5)*, under the K. Let $\mathsf{PA}_K \in \mathbb{C} \to \mathbb{S}^\sharp$ be the result of the reachability-based algorithm (Definition 5) 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 1 and Lemma 2, where the proof of Lemma 1 is immediate from the abstract interpretation framework [1, 2] and we omit the proof. We prove Lemma 2 in A.2. □

A.2 Proof of Lemma 2

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 (6) 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 (6) (Lemmas 7 and 8) and PA_K is equivalent to $\mathcal{X}$ (Lemma 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 3.
2. Let V be the set of (c, κ)'s reachable from (ι, ϵ):
$$V = \{(c, \kappa) \mid (\iota, \epsilon) \to^*_K (c, \kappa)\}$$
3. We define a set Ω_v of generators for each abstract value $v \in \mathbb{V}$:
$$\begin{aligned}\Omega_v = (\mathit{if}\ (v = \top_v)\ \mathit{then}\ \{((\iota, \epsilon), x) \mid x \in \mathsf{Var}\}\ \mathit{else}\ \{\}) \\ \cup \{((c, \kappa), x) \mid \mathsf{cmd}(c) = x := e \wedge \mathsf{const}(e) = v\}\end{aligned}$$
4. Finally, using what we have defined so far, we construct $\mathcal{X} \in \mathbb{D}^\sharp = \mathbb{C}_K \to \mathbb{S}^\sharp$:
$$\begin{aligned}\mathcal{X}(c, \kappa) = \mathit{if}\ ((c, \kappa) \notin V)\ \mathit{then}\ \bot \\ \mathit{else}\ \lambda x.\ \bigsqcup\{v \in \mathbb{V} \mid \exists ((c_0, \kappa_0), x_0) \in \Omega_v. \\ (c_0, \kappa_0) \in V\ \wedge \\ ((c_0, \kappa_0), x_0) \hookrightarrow^*_K ((c, \kappa), x)\}\end{aligned}$$

Lemma 7. *The $\mathcal{X}$ is a solution of* (6)*. That is,*

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

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 $\mathcal{X}$,

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

This implies two important facts. First,

$$w \sqsubseteq v \tag{21}$$

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

$$\begin{aligned}&\forall((c_0, \kappa_0), x_0).\\ &\quad ((c_0, \kappa_0) \in V \ \wedge\ ((c_0, \kappa_0), x_0) \hookrightarrow_K ((c, \kappa), x))\\ &\qquad\qquad \implies \mathcal{X}(c_0, \kappa_0)(x_0) \sqsubseteq v.\end{aligned} \tag{22}$$

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 (21) and (22) follows that

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

as desired. □

Lemma 8. *The $\mathcal{X}$ is a lower bound for every solution of* (6). *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

$$\forall(c, \kappa) \in \mathbb{C}_K.\ \mathcal{X}(c, \kappa) \sqsubseteq X(c, \kappa). \tag{23}$$

First, we show that

$$\forall(c, \kappa) \in V.\ X(c, \kappa) \neq \bot. \tag{24}$$

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., (24)), we prove (23). 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 (24). 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

$$\begin{aligned}&((c_0, \kappa_0), x_0) \in \Omega_v \ \wedge\ (c_0, \kappa_0) \in V\\ &\ \wedge\ (\forall 0 \leq i < n.\ ((c_i, \kappa_i), x_i) \hookrightarrow_K ((c_{i+1}, \kappa_{i+1}), x_{i+1}))\\ &\ \wedge\ ((c_n, \kappa_n), x_n) = ((c, \kappa), x).\end{aligned} \tag{25}$$

Note that every (c_i, κ_i) is in V. So, by (24),

$$\forall 0 \leq i \leq n.\ X(c_i, \kappa_i) \neq \bot. \tag{26}$$

We will show that

$$\begin{aligned}&v \sqsubseteq X(c_0, \kappa_0)(x_0)\\ &\ \wedge\ (\forall 0 \leq i < n.\ X(c_i, \kappa_i)(x_i) \sqsubseteq X(c_{i+1}, \kappa_{i+1})(x_{i+1})).\end{aligned} \tag{27}$$

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 (27) is to notice that

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

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

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

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 (27). In this case, we use a general fact that if

$$((c', \kappa'), x') \hookrightarrow_K ((c'', \kappa''), x'') \ \wedge\ X(c', \kappa') \neq \bot, \tag{28}$$

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 (27) follows if we discharge the condition (28) for consecutive elements in the sequence

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

This condition holds because of (25) and (26). □

Lemma 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 (7):

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

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

$$\mathsf{PA}_K(c)(x) = \bigsqcup_{\kappa \in K_0} \mathcal{X}(c,\kappa)(x). \tag{29}$$

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

$$V_L = \{v \in \mathbb{V} \mid \exists (c_0, x_0) \in \Theta_v.\ (c_0, x_0) \hookrightarrow_K^\dagger (c, x)\}. \tag{30}$$

The right hand side of the equation in (29) is the join of the set

$$\begin{aligned} V_R = \{v \in \mathbb{V} \mid \exists \kappa \in K_0.\ &\exists((c_0,\kappa_0),x_0) \in \Omega_v. \\ &(c_0,\kappa_0) \in V \ \wedge \\ &((c_0,\kappa_0),x_0) \hookrightarrow_K^* ((c,\kappa),x)\}. \end{aligned} \tag{31}$$

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. $\square$