Noname manuscript No.
(will be inserted by the editor)

Learning Analysis Strategies for Octagon and Context Sensitivity from Labeled Data Generated by Static Analyses

Kihong Heo · Hakjoo Oh · Hongseok Yang

Received: date / Accepted: date

Abstract We present a method for automatically learning an effective strategy for clustering variables for the Octagon analysis from a given codebase. This learned strategy works as a preprocessor of Octagon. Given a program to be analyzed, the strategy is first applied to the program and clusters variables in it. We then run a partial variant of the Octagon analysis that tracks relationships among variables within the same cluster, but not across different clusters. The notable aspect of our learning method is that although the method is based on supervised learning, it does not require manually-labeled data. The method does not ask human to indicate which pairs of program variables in the given codebase should be tracked. Instead it uses the impact pre-analysis for Octagon from our previous work and automatically labels variable pairs in the codebase as positive or negative. We implemented our method on top of a static buffer-overflow detector for C programs and tested it against open source benchmarks. Our experiments show that the partial Octagon analysis with the learned strategy scales up to 100KLOC and is 33x faster than the one with the impact pre-analysis (which itself is significantly faster than the original Octagon analysis), while increasing false alarms by only 2%. The general idea behind our method is applicable to other types of static analyses as well. We demonstrate that our method is also effective to learn a strategy for context-sensitivity of interval analysis.

This work was carried out while Heo was at Seoul National University and Yang was at the University of Oxford.

Kihong Heo
E-mail: kheo@cis.upenn.edu

Hakjoo Oh
E-mail: hakjoo_oh@korea.ac.kr

Hongseok Yang
E-mail: hongseok.yang@kaist.ac.kr

Keywords Static Analysis · Machine Learning · Relational Analysis · Context-sensitivity

1 Introduction

Relational program analyses track sophisticated relationships among program variables and enable the automatic verification of complex properties of programs [3, 9]. However, the computational costs of various operations of these analyses are high so that vanilla implementations of the analyses do not scale even to moderate-sized programs. For example, transfer functions of the Octagon analysis [9] have a cubic worst-case time complexity in the number of program variables, which makes it impossible to analyze large programs.

In this paper, we consider one of the most popular optimizations used by practical relational program analyses, called variable clustering [9, 16, 28, 1]. Given a program, an analyzer with this optimization forms multiple relatively-small subsets of variables, called variable clusters or clusters. Then, it limits the tracked information to the relationships among variables within each cluster, not across those clusters. So far strategies based on simple syntactic or semantic criteria have been used for clustering variables for a given program, but they are not satisfactory. They are limited to a specific class of target programs [28, 1] or employ a pre-analysis that is cheaper than a full relational analysis but frequently takes order-of-magnitude more time than the non-relational analysis for medium-sized programs [16].

In this paper, we propose a new method for automatically learning a variable-clustering strategy for the Octagon analysis from a given codebase. When applied to a program, the learned strategy represents each pair of variables (x_i, x_j) in the program by a boolean vector, and maps such a vector to $\oplus$ or $\ominus$, where $\oplus$ signifies the importance of tracking the relationship between x_i and x_j. If we view such $\oplus$-marked (x_i, x_j) as an edge of a graph, the variant of Octagon in this paper decides to track the relationship between variables x and y only when there is a path from x to y in the graph. According to our experiments, running this strategy for all variable pairs is quick and results in a good clustering of variables, which makes the variant of Octagon achieve performance comparable to the non-relational Interval analysis while enjoying the accuracy of the original Octagon in many cases.

The most important aspect of our learning method is the automatic provision of labeled data. Although the method is essentially an instance of supervised learning, it does not require the common unpleasant involvement of humans in supervised learning, namely, labeling. Our method takes a codebase consisting of typical programs of small-to-medium size, and automatically generates labels for pairs of variables in those programs by using the impact pre-analysis from our previous work [16], which estimates the impact of tracking relationships among variables by Octagon on proving queries in given programs. Our method precisely labels a pair of program variables with $\oplus$ when the pre-analysis says that the pair should be tracked. Because this

learning occurs offline, we can bear the cost of the pre-analysis, which is still significantly lower than the cost of the full Octagon analysis. Once labeled data are generated, our method runs an off-the-shelf classification algorithm, such as decision-tree inference [10], for inferring a classifier for those labeled data. This classifier is used to map vector representations of variable pairs to $\oplus$ or $\ominus$. Conceptually, the inferred classifier is a further approximation of the pre-analysis, which is found automatically from a given codebase.

The experimental results show that our method results in the learning of a cost-effective variable-clustering strategy. We implemented our learning method on top of a static buffer overflow detector for C programs and tested against open source benchmarks. In the experiments, our analysis with the learned variable-clustering strategy scales up to 100KLOC within the two times of the analysis cost of the Interval analysis. This corresponds to the 33x speed-up of the selective relational analysis based on the impact pre-analysis [16] (which was already significantly faster than the original Octagon analysis). The price of speed-up was mere 2% increase of false alarms.

The general idea of this approach is applicable to learn a strategy for other types of sensitivities in static analysis. To demonstrate it, we designed a partially context-sensitive interval analysis by following the same idea. The experimental results show that our method is also effective to learn a strategy for context-sensitivity.

We summarize the contributions of this paper below:

1. We propose a method for automatically learning an effective strategy for variable-clustering for the Octagon analysis from a given codebase. The method infers a function that decides, for a program P and a pair of variables (x, y) in P, whether tracking the relationship between x and y is important. The learned strategy uses this function to cluster variables in a given program.
2. We show how to automatically generate labeled data from a given codebase that are needed for learning. Our key idea is to generate such data using the impact pre-analysis for Octagon from [16]. This use of the pre-analysis means that our learning step is just the process of finding a further approximation of the pre-analysis, which avoids expensive computations of the pre-analysis but keeps its important estimations.
3. We experimentally show the effectiveness of our learning method using a realistic static analyzer for full C and open source benchmarks. Our variant of Octagon with the learned strategy is 33x faster than the selective relational analysis based on the impact pre-analysis [16] while increasing false alarms by only 2%.
4. We show that the general idea behind our method is also applicable to learning context-sensitivity of interval analysis.

This paper is an extended version of [6]. The major extension is the addition of a new instance for context-sensitivity; we give detailed descriptions on applying our method to context-sensitive interval analysis (Sections 5 and 6) and provide experimental results (Section 7).

```
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  }
```

Fig. 1 Example Program

2 Informal Explanation

We informally explain our approach with a partially relational Octagon analysis using the program in Figure 1. The program contains two queries about the relationships between `i` and variables `a`, `b` inside the loop. The first query `i` < `a` is always true because the loop condition ensures `i` < `b` and variables `a` and `b` have the same value throughout the loop. The second query `i` < `c`, on the other hand, may become false because `c` is set to an unknown input at line 2.

2.1 Octagon Analysis with Variable Clustering

The Octagon analysis [9] discovers program invariants strong enough to prove the first query in our example. At each program point it infers an invariant of the form

$$\Big(\bigwedge_{ij} L_{ij} \leq x_j + x_i \leq U_{ij}\Big) \wedge \Big(\bigwedge_{ij} L'_{ij} \leq x_j - x_i \leq U'_{ij}\Big)$$

for $L_{ij}, L'_{ij} \in \mathbb{Z} \cup \{-\infty\}$ and $U_{ij}, U'_{ij} \in \mathbb{Z} \cup \{\infty\}$. In particular, at the first query of our program, the analysis infers the following invariant, which we present in the usual matrix form:

	a	−a	b	−b	c	−c	i	−i
a	0	∞	0	∞	∞	∞	−1	∞
−a	∞	0	∞	0	∞	∞	∞	∞
b	0	∞	0	∞	∞	∞	−1	∞
−b	∞	0	∞	0	∞	∞	∞	∞
c	∞	∞	∞	∞	0	∞	∞	∞
−c	∞	∞	∞	∞	∞	0	∞	∞
i	∞	∞	∞	∞	∞	∞	0	∞
−i	∞	−1	∞	−1	∞	∞	∞	0

(1)

The ij-th entry m_{ij} of this matrix means an upper bound $e_j - e_i \leq m_{ij}$, where e_j and e_i are expressions associated with the j-th column and the i-th row of the matrix respectively and they are variables with or without the minus sign. The matrix records -1 and ∞ as upper bounds for `i` − `a` and `i` − `c`, respectively. Note that these bounds imply the first query, but not the second.

In practice the Octagon analysis is rarely used without further optimization, because it usually spends a large amount of computational resources for discovering unnecessary relationships between program variables, which do

not contribute to proving given queries. In our example, the analysis tracks the relationship between `c` and `i`, although it does not help prove any of the queries.

A standard approach for addressing this inefficiency is to form subsets of variables, called variable clusters or clusters. According to a pre-defined clustering strategy, the analysis tracks the relationships between only those variables within the same cluster, not across clusters. In our example, this approach would form two clusters $\{\texttt{a}, \texttt{b}, \texttt{i}\}$ and $\{\texttt{c}\}$ and prevent the Octagon analysis from tracking the unnecessary relationships between `c` and the other variables. The success of the approach lies in finding a good strategy that is able to find effective clusters for a given program. This is possible as demonstrated in the several previous work [16,28,1], but it is highly nontrivial and often requires a large amount of trial and error of analysis designers.

Our goal is to develop a method for automatically learning a good variable-clustering strategy for a target class of programs. By "a target class of programs", we mean that training and test programs are not totally different from the viewpoint of a particular program analyzer. For instance, both training and test programs may be GNU open-source programs written in C, as in our experiments, so that they follow some common programming styles. This assumption makes learning possible. The literature of machine learning often makes a similar assumption that training and test data are drawn from the same but unknown distribution in terms of general features.

This automatic learning happens offline with a collection of typical sample programs from the target class, and the learned strategy is later applied to any programs in the class, most of which are not used during learning. We want the learned strategy to form relatively-small variable clusters so as to lower the analysis cost and, at the same time, to put a pair of variables in the same cluster if tracking their relationship by Octagon is important for proving given queries. For instance, such a strategy would cluster variables of our example program into two groups $\{\texttt{a}, \texttt{b}, \texttt{i}\}$ and $\{\texttt{c}\}$, and make Octagon compute the following smaller matrix at the first query:

	a	−a	b	−b	i	−i
a	0	∞	0	∞	−1	∞
−a	∞	0	∞	0	∞	∞
b	0	∞	0	∞	−1	∞
−b	∞	∞	∞	0	∞	∞
i	∞	∞	∞	∞	0	∞
−i	∞	−1	∞	−1	∞	∞

(2)

2.2 Automatic Learning of a Variable-Clustering Strategy

In this paper we will present a method for learning a variable-clustering strategy. Using a given codebase, it infers a function $\mathcal{F}$ that maps a tuple $(P, (x, y))$ of a program P and variables x, y in P to $\oplus$ and $\ominus$. The output $\oplus$ here means that tracking the relationship between x and y is likely to be important for proving queries. The inferred $\mathcal{F}$ guides our variant of the Octagon analysis.

Given a program P, our analysis applies $\mathcal{F}$ to every pair of variables in P, and computes the finest partition of variables that puts every pair (x, y) with the $\oplus$ mark in the same group. Then, it analyzes the program P by tracking relationships between variables within each group in the partition, but not across groups.

Our method for learning takes a codebase that consists of typical programs in the intended application of the analysis. Then, it automatically synthesizes the above function $\mathcal{F}$ in two steps. First, it generates labeled data automatically from programs in the codebase by using the impact pre-analysis for Octagon from our previous work [16]. This is the most salient aspect of our approach; in a similar supervised-learning task, such labeled data are typically constructed manually, and avoiding this expensive manual labelling process is considered a big challenge for supervised learning. Next, our approach converts labeled data to boolean vectors marked with $\oplus$ or $\ominus$, and runs an off-the-shelf supervised learning algorithm to infer a classifier, which is used to define $\mathcal{F}$.

Note that The full Octagon analysis is not appropriate for labeling. In our experiments, it scales only up to 13KLOC and generates 1 million labeled variable pairs for training data. By contrast, the pre-analysis scales up to 110KLOC and generates 258 million labeled variable pairs for training data. Using the former (much smaller) training data results in a poor classifier that selects too many variable pairs, and as a result, the resulting partial Octagon analysis does not scales beyond 50KLOC. Note that our pre-analysis is sufficiently precise; it can select 98% of queries that are provable by the full Octagon analysis [16] on average.

Automatic Generation of Labeled Data Labeled data in our case are a collection of triples $(P, (x, y), L)$ where P is a program, (x, y) is a pair of variables in P, and $L \in \{\oplus, \ominus\}$ is a label that indicates whether tracking the relationship between x and y is important. We generate such labeled data automatically from the programs $P_1, \ldots, P_N$ in the given codebase.

The key idea is to use the impact pre-analysis for Octagon [16], and to convert the results of this pre-analysis to labeled data. Just like the Octagon analysis, this pre-analysis tracks the relationships between variables, but it aggressively abstracts any numerical information so as to achieve better scalability than Octagon. The goal of the pre-analysis is to identify, as much as possible, the case that Octagon would give a precise bound for $\pm x \pm y$, without running Octagon itself. As in Octagon, the pre-analysis computes a matrix with rows and columns for variables with or without the minus sign, but this matrix $m^\sharp$ contains $\bigstar$ or $\top$, instead of any numerical values. For instance, when applied to our example program, the pre-analysis would infer the following matrix at the first query:

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

(3)

Each entry of this matrix stores the pre-analysis's (highly precise on the positive side) prediction on whether Octagon would put a *finite* upper bound at the corresponding entry of its matrix at the same program point. $\bigstar$ means likely, and $\top$ unlikely. For instance, the above matrix contains $\bigstar$ for the entries for $\texttt{i} - \texttt{b}$ and $\texttt{b} - \texttt{a}$, and this means that Octagon is likely to infer finite (thus informative) upper bounds of $\texttt{i} - \texttt{b}$ and $\texttt{b} - \texttt{a}$. In fact, this predication is correct because the actual upper bounds inferred by Octagon are -1 and 0, as can be seen in (1).

We convert the results of the impact pre-analysis to labeled data as follows. For every program P in the given codebase, we first collect all queries $Q = \{q_1, \dots, q_k\}$ that express legal array accesses or the success of assert statements in terms of upper bounds on $\pm x \pm y$ for some variables x, y. Next, we filter out queries $q_i \in Q$ such that the upper bounds associated with q_i are not predicted to be finite by the pre-analysis. Intuitively, the remaining queries are the ones that are likely to be proved by Octagon according to the prediction of the pre-analysis. Then, for all remaining queries $q'_1, \dots, q'_l$, we collect the results $m^\sharp_1, \dots, m^\sharp_l$ of the pre-analysis at these queries, and generate the following labeled data:

$$\begin{aligned} &\mathcal{D}_P = \{(P, (x, y), L) \mid \\ &\quad L = \oplus \iff \text{at least one of the entries of some } m_i \text{ for } \pm x \pm y \text{ has } \bigstar\}. \end{aligned}$$

Notice that we mark (x, y) with $\oplus$ if tracking the relationship between x and y is useful for some query q'_i. An obvious alternative is to replace some by all, but we found that this alternative led to the worse performance in our experiments.[1] This generation process is applied for all programs $P_1, \dots, P_N$ in the codebase, and results in the following labeled data: $\mathcal{D} = \bigcup_{1 \le i \le N} \mathcal{D}_{P_i}$. In our example program, if the results of the pre-analysis at both queries are the same matrix in (3), our approach picks only the first matrix because the pre-analysis predicts a finite upper bound only for the first query, and it produces the following labeled data from the first matrix:

$$\{(P, t, \oplus) \mid t \in T\} \cup \{(P, t, \ominus) \mid t \notin T\}$$

where $T = \{(\texttt{a}, \texttt{b}), (\texttt{b}, \texttt{a}), (\texttt{a}, \texttt{i}), (\texttt{i}, \texttt{a}), (\texttt{b}, \texttt{i}), (\texttt{i}, \texttt{b}), (\texttt{a}, \texttt{a}), (\texttt{b}, \texttt{b}), (\texttt{c}, \texttt{c}), (\texttt{i}, \texttt{i})\}$.

[1] Because the pre-analysis uses $\bigstar$ cautiously, only a small portion of variable pairs is marked with $\oplus$ (that is, $5864/258,165,546$) in our experiments. Replacing "some" by "all" reduces this portion by half ($2230/258,165,546$) and makes the learning task more difficult.

```
1  int id1(int x){ return x; }
2  int id2(int x){ return id1(x); }
3  int id3(int x){ return x; }
4  int id4(int x){ return id3(x); }
5
6  void main(){
7    a = id2(1);
8    b = id2(input());
9    assert(a > 0);      // Query 1
10   assert(b > 0);      // Query 2
11
12   c = id4(input());
13   d = id4(input());
14   assert(c > 0);      // Query 3
15   assert(d > 0);      // Query 4
16 }
```

Fig. 2 Example Program

Application of an Off-the-shelf Classification Algorithm Once we generate labeled data $\mathcal{D}$, we represent each triple in $\mathcal{D}$ as a vector of $\{0, 1\}$ labeled with $\oplus$ or $\ominus$, and apply an off-the-shelf classification algorithm, such as decision-tree inference [10].

The vector representation of each triple in $\mathcal{D}$ is based on a set of so called features, which are syntactic or semantic properties of a variable pair (x, y) under a program P. Formally, a feature f maps such $(P, (x, y))$ to 0 or 1. For instance, f may check whether the variables x and y appear together in an assignment of the form $x = y + c$ in P, or it may test whether x or y is a global variable. Table 1 lists all the features that we designed and used for our variant of the Octagon analysis. Let us denote these features and results of applying them using the following symbols:

$$\mathbf{f} = \{f_1, \dots, f_m\}, \quad \mathbf{f}(P, (x, y)) = \big(f_1(P, (x, y)), \dots, f_m(P, (x, y))\big) \in \{0, 1\}^m.$$

The vector representation of triples in $\mathcal{D}$ is the following set:

$$\mathcal{V} = \{(\mathbf{f}(P, (x, y)), L) \mid (P, (x, y), L) \in \mathcal{D}\} \in \wp(\{0, 1\}^m \times \{\oplus, \ominus\})$$

We apply an off-the-self classification algorithm to the set. In our experiments, the algorithm for learning a decision tree gave the best classifier for our variant of the Octagon analysis.

2.3 Application to Context-sensitivity

The general principle of our learning approach is also applicable to other partially sensitive analyses. Figure 2 shows an example program that requires a context-sensitive analysis. The program contains four queries; the first query always holds but the others are not always true because of the unknown inputs.

If we use the Interval analysis for the example program, both of context-insensitive and uniformly context-sensitive analysis are not an appropriate

choice. The context-insensitive analysis cannot prove any queries. Since the analysis merges all inputs of each function (`id1` and `id2`), both `x` and `y` at line 1 and 3 have $[-\infty, +\infty]$. An uniformly context-sensitive analysis such as 2-CFA [22] can prove the first query by differentiating all the calling contexts separately:

$$2 \cdot 7 \qquad 2 \cdot 8 \qquad 4 \cdot 12 \qquad 4 \cdot 13$$

However, a uniformly context-sensitive analysis is not cost-effective. The context-sensitivity on `id3` and `id4` does not help to prove any query in the example.

Instead, our learning method derives an effective strategy for partial context-sensitivity. Like the partial Octagon case, we also infer a function $\mathcal{F}$ that maps a tuple (P, ψ) of a program P and function ψ in P to $\oplus$ and $\ominus$. $\oplus$ means that context sensitivity on the function is likely to be important for proving queries. According to the result of $\mathcal{F}$, we derive a partial context-sensitivity. For example, the learned strategy would construct the following calling contexts for the first query in the example:

$$2 \cdot 7 \qquad 2 \cdot 8 \qquad \text{all the other contexts}$$

Likewise, we automatically generate labeled data form the given codebase and the fully context-sensitive impact pre-analysis for context-sensitivity [16]. The pre-analysis approximates the interval analysis using the following abstract domain:

$$\{\bot\} \cup (\mathit{Var} \rightarrow \{\top, \bigstar\})$$

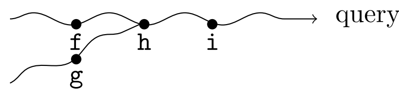
where $\bigstar$ means positive intervals (i.e., $[l, u]$ where $0 \leq l \leq u$) and $\top$ means all intervals. For example, the following table shows the summary of the main analysis and the pre-analysis on the example program. The second and third columns represent the return values of each analysis following the corresponding call chains:

Context	Main analysis	Pre-analysis
$2 \cdot 7$	$[1, 1]$	$\bigstar$
$2 \cdot 8$	$[-\infty, +\infty]$	$\top$
$4 \cdot 12$	$[-\infty, +\infty]$	$\top$
$4 \cdot 13$	$[-\infty, +\infty]$	$\top$

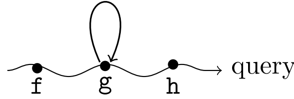
The pre-analysis precisely estimates the fact that the main analysis can have the precise result along the calling context $2 \cdot 7$. From the pre-analysis result, we generate labeled data.

$$\mathcal{D}_P = \{(P, \psi, L) \mid L = \oplus \iff \psi \text{ is on a dependency path of a query that has } \bigstar\}$$

For example, if a dependency path for a selected query looks as follows:



Then function `f`, `g`, `h`, and `i` are positive examples. However, if a slice involves a recursive call as follows:



we exclude the query since otherwise, we need infinitely many different calling contexts.

After we generate labeled data, we follow the same process in the partial Octagon case in Section 2.2. We represent each tuple in the training data as a feature vector and derive a classifier using an off-the-shelf machine learning algorithm.

3 Octagon Analysis with Variable Clustering

In this section, we describe a variant of the Octagon analysis that takes not just a program to be analyzed but also clusters of variables in the program. Such clusters are computed according to some strategy before the analysis is run. Given a program and variable clusters, this variant Octagon analysis infers relationships between variables within the same cluster but not across different clusters. Section 4 presents our method for automatically learning a good strategy for forming such variable clusters.

3.1 Programs

A program is represented by a control-flow graph $(\mathbb{C}, \rightarrow)$, where $\mathbb{C}$ is the set of program points and $(\rightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ denotes the control flows of the program. Let $\mathit{Var}_n = \{x_1, \ldots, x_n\}$ be the set of variables. Each program point $c \in \mathbb{C}$ has a primitive command working with these variables. When presenting the formal setting and our results, we mostly assume the following collection of simple primitive commands:

$$\mathit{cmd} \ ::= \ x = k \mid x = y + k \mid x = \,?$$

where x, y are program variables, $k \in \mathbb{Z}$ is an integer, and $x = \,?$ is an assignment of some nondeterministically-chosen integer to x. The Octagon analysis is able to handle the first two kinds of commands precisely. The last command is usually an outcome of a preprocessor that replaces a complex assignment such as non-linear assignment $x = y * y + 1$ (which cannot be analyzed accurately by the Octagon analysis) by this overapproximating non-deterministic assignment.

3.2 Octagon Analysis

We briefly review the Octagon analysis in [9]. Let $\mathit{Var}_n = \{x_1, \ldots, x_n\}$ be the set of variables that appear in a program to be analyzed. The analysis aims at

$$[\![x_i = k]\!]m = m' \text{ when } m'_{pq} = \begin{cases} -2k & p = 2i-1 \wedge q = 2i \\ 2k & p = 2i \wedge q = 2i-1 \\ ([\![x_i = ?]\!]m)_{pq} & \mathit{otherwise} \end{cases}$$

$$([\![x_i = x_i + k]\!]m)_{pq} = \begin{cases} m_{pq} - k & (p = 2i-1 \wedge q \notin \{2i-1, 2i\}) \vee (p \notin \{2i-1, 2i\} \wedge q = 2i) \\ m_{pq} + k & (p = 2i \wedge q \notin \{2i-1, 2i\}) \vee (p \notin \{2i-1, 2i\} \wedge q = 2i-1) \\ m_{pq} - 2k & p = 2i-1 \wedge q = 2i \\ m_{pq} + 2k & p = 2i \wedge q = 2i-1 \\ m_{pq} & \mathit{otherwise} \end{cases}$$

$$[\![x_i = x_j + k]\!]m = m' \text{ when } m'_{pq} = \begin{cases} -k & p = 2i-1 \wedge q = 2j-1 \\ -k & p = 2j \wedge q = 2i \\ k & p = 2j-1 \wedge q = 2i-1 \\ k & p = 2i \wedge q = 2j \\ ([\![x_i = ?]\!]m)_{pq} & \mathit{otherwise} \end{cases}$$

$$[\![x_i = ?]\!]m = \bot \text{ when } m^\bullet = \bot$$

$$[\![x_i = ?]\!]m = m' \text{ when } m^\bullet \neq \bot \text{ and } m'_{pq} = \begin{cases} \infty & p \in \{2i-1, 2i\} \wedge q \notin \{2i-1, 2i\} \\ \infty & p \notin \{2i-1, 2i\} \wedge q \in \{2i-1, 2i\} \\ 0 & p = q = 2i-1 \vee p = q = 2i \\ (m^\bullet)_{pq} & \mathit{otherwise} \end{cases}$$

Fig. 3 Abstract semantics of some primitive commands in the Octagon analysis. We show the case that the input m is not $\bot$; the abstract semantics always maps $\bot$ to $\bot$.

finding the upper and lower bounds of expressions of the forms x_i, $x_i + x_j$ and $x_i - x_j$ for variables $x_i, x_j \in \mathit{Var}_n$. The analysis represents these bounds as a $(2n \times 2n)$ matrix m of values in $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$, which means the following constraint:

$$\bigwedge_{(1 \le i,j \le n)} \bigwedge_{(k,l \in \{0,1\})} ((-1)^{l+1} x_j - (-1)^{k+1} x_i) \le m_{(2i-k)(2j-l)}$$

The abstract domain $\mathbb{O}$ of the Octagon analysis consists of all those matrices and $\bot$, and uses the following pointwise order: for $m, m' \in \mathbb{O}$,

$$m \sqsubseteq m' \iff (m = \bot) \vee (m \neq \bot \wedge m' \neq \bot \wedge \forall 1 \le i,j \le 2n.\ (m_{ij} \le m'_{ij})).$$

This domain is a complete lattice $(\mathbb{O}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where $\top$ is the matrix containing only ∞ and $\sqcup$ and $\sqcap$ are defined pointwise. The details of the lattice structure can be found in [9].

Usually multiple abstract elements of $\mathbb{O}$ mean constraints with the same set of solutions. If we fix a set S of solutions and collect in the set M all the abstract elements with S as their solutions, the set M always contains the least element according to the $\sqsubseteq$ order. There is a cubic-time algorithm for computing this least element from any $m \in M$. We write $m^\bullet$ to denote the result of this algorithm, and call it strong closure of m.

The abstract semantics of primitive commands $[\![\mathit{cmd}]\!] : \mathbb{O} \to \mathbb{O}$ is defined in Figure 3. The effects of the first two assignments in the concrete semantics can be tracked precisely using abstract elements of Octagon. The abstract semantics of these assignments do such tracking. $[\![x_i = ?]\!]m$ in the last case

computes the strong closure of m and forgets any bounds involving x_i in the resulting abstract element $m^\bullet$. The analysis computes a pre-fixpoint of the semantic function $F : (\mathbb{C} \to \mathbb{O}) \to (\mathbb{C} \to \mathbb{O})$ (i.e., X_I with $F(X_I)(c) \sqsubseteq X_I(c)$ for all $c \in \mathbb{C}$):

$$F(X)(c) = [\![cmd(c)]\!](\bigsqcup_{c' \to c} X(c'))$$

where $cmd(c)$ is the primitive command associated with the program point c.

3.3 Variable Clustering and Partial Octagon Analysis

We use a program analysis that performs the Octagon analysis only partially. This variant of Octagon is similar to those in [9,16]. This partial Octagon analysis takes a collection Π of clusters of variables, which are subsets π of variables in Var_n such that $\bigcup_{\pi \in \Pi} \pi = Var_n$. Each $\pi \in \Pi$ specifies a variable cluster and instructs the analysis to track relationships between variables in π. Given such a collection Π, the partial Octagon analysis analyzes the program using the complete lattice $(\mathbb{O}_\Pi, \sqsubseteq_\Pi, \bot_\Pi, \top_\Pi, \sqcup_\Pi, \sqcap_\Pi)$ where

$$\mathbb{O}_\Pi = \prod_{\pi \in \Pi} \mathbb{O}_\pi \qquad (\mathbb{O}_\pi \text{ is the lattice of Octagon for variables in } \pi).$$

That is, $\mathbb{O}_\Pi$ consists of families $\{m_\pi\}_{\pi \in \Pi}$ such that each m_π is an abstract element of Octagon used for variables in π, and all lattice operations of $\mathbb{O}_\Pi$ are the pointwise extensions of those of Octagon. For the example in Section 2, if we use $\Pi = \{\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{i}\}\}$, the partial Octagon analysis uses the same domain as Octagon's. But if $\Pi = \{\{\mathtt{a}, \mathtt{b}, \mathtt{i}\}, \{\mathtt{c}\}\}$, the analysis uses the product of two smaller abstract domains, one for $\{\mathtt{a}, \mathtt{b}, \mathtt{i}\}$ and the other for $\{\mathtt{c}\}$.

The partial Octagon analysis computes a pre-fixpoint of the following F_Π:

$$F_\Pi : (\mathbb{C} \to \mathbb{O}_\Pi) \to (\mathbb{C} \to \mathbb{O}_\Pi)$$
$$F_\Pi(X)(c) = [\![cmd(c)]\!]_\Pi(\bigsqcup_{c' \to c} X(c')).$$

Here the abstract semantics $[\![cmd(c)]\!]_\Pi : \mathbb{O}_\Pi \to \mathbb{O}_\Pi$ of the command c is defined in terms of Octagon's:

$$([\![x_i = ?]\!]_\Pi po)_\pi = \begin{cases} [\![x_i = ?]\!](po_\pi) & x_i \in \pi \\ po_\pi & otherwise \end{cases}$$
$$([\![x_i = k]\!]_\Pi po)_\pi = \begin{cases} [\![x_i = k]\!](po_\pi) & x_i \in \pi \\ po_\pi & otherwise \end{cases}$$
$$([\![x_i = x_j + k]\!]_\Pi po)_\pi = \begin{cases} [\![x_i = x_j + k]\!](po_\pi) & x_i, x_j \in \pi \\ [\![x_i = ?]\!](po_\pi) & otherwise \end{cases}$$

The abstract semantics of a command updates the component of an input abstract state for a cluster π if the command changes any variable in the cluster; otherwise, it keeps the component. The update is done according to

the abstract semantics of Octagon. Notice that the abstract semantics of $x_i = x_j + k$ does not track the relationship $x_j - x_i = k$ in the π component when $x_i \in \pi$ but $x_j \notin \pi$. Giving up such relationships makes this partial analysis perform faster than the original Octagon analysis.

4 Learning a Strategy for Clustering Variables

The success of the partial Octagon analysis lies in choosing good clusters of variables for a given program. Ideally each cluster of variable should be relatively small, but if tracking the relationship between variables x_i and x_j is important, some cluster should contain both x_i and x_j. In this section, we present a method for learning a strategy that chooses such clusters. Our method takes as input a collection of programs, which reflects a typical usage scenario of the partial Octagon analysis. It then automatically learns a strategy from this collection.

In the section we assume that our method is given $\{P_1, \ldots, P_N\}$, and we let

$$\mathcal{P} = \{(P_1, Q_1), \ldots, (P_N, Q_N)\},$$

where Q_i means a set of queries in P_i. It consists of pairs (c, p) of a program point c of P_i and a predicate p on variables of P_i, where the predicate expresses an upper bound on variables or their differences, such as $x_i - x_j \leq 1$. Another notation that we adopt is Var_P for each program P, which means the set of variables appearing in P.

4.1 Automatic Generation of Labeled Data

The first step of our method is to generate labeled data from the given collection $\mathcal{P}$ of programs and queries. In theory the generation of this labeled data can be done by running the full Octagon analysis. For every (P_i, Q_i) in $\mathcal{P}$, we run the Octagon analysis for P_i, and collect queries in Q_i that are proved by the analysis. Then, we label a pair of variable (x_j, x_k) in P_i with $\oplus$ if (i) a nontrivial[2] upper or lower bound (x_i, x_k) is computed by the analysis at some program point c in P_i and (ii) the proof of some query by the analysis depends on this nontrivial upper bound. Otherwise, we label the pair with $\ominus$. The main problem with this approach is that we cannot analyze all the programs in $\mathcal{P}$ with Octagon because of the scalability issue of Octagon.

In order to lessen this scalability issue, we instead run the impact pre-analysis for Octagon from our previous work [16], and convert its results to labeled data. Although this pre-analysis is not as cheap as the Interval analysis, it scales far better than Octagon and enables us to generate labeled data from a wide range of programs. Our learning method then uses the generated data to find a strategy for clustering variables. The found strategy can be viewed

[2] By nontrivial, we mean finite bounds that are neither ∞ nor $-\infty$.

as an approximation of this pre-analysis that scales as well as the Interval analysis.

Impact Pre-analysis We review the impact pre-analysis from [16], which aims at quickly predicting the results of running the Octagon analysis on a given program P. Let $n = |\mathit{Var}_P|$, the number of variables in P. At each program point c of P, the pre-analysis computes a $(2n \times 2n)$ matrix $m^\sharp$ with entries in $\{\bigstar, \top\}$. Intuitively, such a matrix $m^\sharp$ records which entries of the matrix m computed by Octagon are likely to contain nontrivial bounds. The pre-analysis over-approximates the main analysis. As a result, if the ij-th entry of $m^\sharp$ is $\bigstar$, the ij-th entry of m is *always* equal to non-∞ according to the prediction of the pre-analysis. The pre-analysis does not make similar prediction for entries of $m^\sharp$ with $\top$. Such entries should be understood as the absence of information.

The pre-analysis uses a complete lattice $(\mathbb{O}^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \top^\sharp, \sqcup^\sharp, \sqcap^\sharp)$ where $\mathbb{O}^\sharp$ consists of $(2n \times 2n)$ matrices of values in $\{\bigstar, \top\}$, the order $\sqsubseteq^\sharp$ is the pointwise extension of the total order $\bigstar \sqsubseteq \top$, and all the other lattice operations are defined pointwise. There exists a Galois connection between The domain of sets of octagons $\wp(\mathbb{O})$ and $\mathbb{O}^\sharp$. $\wp(\mathbb{O})$ is $\cup$-complete lattice:

$$(\wp(\mathbb{O}), \subseteq, \emptyset, \mathbb{O}, \cup, \cap)$$

Abstract state $m^\sharp \in \mathbb{O}^\sharp$ denotes a set of octagons that is characterized by the following Galois connection:

$$\wp(\mathbb{O}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{O}^\sharp$$

$$\big(\alpha(M)\big)_{ij} = \begin{cases} \bigstar \text{ if } +\infty \notin \{m_{ij} \mid m \in M \setminus \{\bot\}\}, \\ \top \text{ otherwise;} \end{cases}$$
$$\gamma(m^\sharp) = \{\bot\} \cup \{m \mid m \neq \bot \wedge \forall i,j.\ (m^\sharp_{ij} = \bigstar \implies m_{ij} \neq +\infty)\}.$$

Lemma 1 (Galois connection) *(α, γ) forms a Galois connection:*

$$\forall M \in \wp(\mathbb{O}), m^\sharp \in \mathbb{O}^\sharp.\ \alpha(M) \sqsubseteq m^\sharp \iff M \subseteq \gamma(m^\sharp).$$

Proof It is enough to prove the following:

$$\alpha(M) \sqsubseteq m^\sharp \iff M \setminus \{\bot\} \subseteq \gamma(m^\sharp) \setminus \{\bot\}.$$

$$[\![x_i = k]\!]^\sharp m^\sharp = m_1^\sharp \text{ when } (m_1^\sharp)_{pq} = \begin{cases} \bigstar & p = 2i-1 \land q = 2i \\ \bigstar & p = 2i \land q = 2i-1 \\ \left([\![x_i = ?]\!]^\sharp m^\sharp\right)_{pq} & \textit{otherwise} \end{cases}$$

$$\left([\![x_i = x_i + k_{>0}]\!]^\sharp m^\sharp\right)_{pq} = \begin{cases} \bigstar & (p = 2i-1 \land q \notin \{2i-1, 2i\}) \lor (p \notin \{2i-1, 2i\} \land q = 2i) \\ \top & (p = 2i \land q \notin \{2i-1, 2i\}) \lor (p \notin \{2i-1, 2i\} \land q = 2i-1) \\ \bigstar & p = 2i-1 \land q = 2i \\ \top & p = 2i \land q = 2i-1 \\ m^\sharp_{pq} & \textit{otherwise} \end{cases}$$

$$[\![x_i = x_i + k_{\leq 0}]\!]^\sharp m^\sharp = m^\sharp$$

$$[\![x_i = x_j + k]\!]^\sharp m^\sharp = m_1^\sharp \text{ when } (m_1^\sharp)_{pq} = \begin{cases} \bigstar & p = 2i-1 \land q = 2j-1 \\ \bigstar & p = 2j \land q = 2i \\ \bigstar & p = 2j-1 \land q = 2i-1 \\ \bigstar & p = 2i \land q = 2j \\ \left([\![x_i = ?]\!]^\sharp m^\sharp\right)_{pq} & \textit{otherwise} \end{cases}$$

$$[\![x_i = ?]\!]^\sharp m^\sharp = m_1^\sharp \text{ when } (m_1^\sharp)_{pq} = \begin{cases} \top & p \in \{2i-1, 2i\} \land q \notin \{2i-1, 2i\} \\ \top & p \notin \{2i-1, 2i\} \land q \in \{2i-1, 2i\} \\ \bigstar & p = q = 2i-1 \lor p = q = 2i \\ ((m^\sharp)^\bullet)_{pq} & \textit{otherwise} \end{cases}$$

Fig. 4 Abstract semantics of some primitive commands in the impact pre-analysis. $k_{>0}$ and $k_{\leq 0}$ present positive and non-positive integers, respectively.

The following derivation shows this equivalence.

$$\begin{aligned}
&\alpha(M) \sqsubseteq m^\sharp \\
&\iff \forall i, j.\ \big((\alpha(M))_{ij} \sqsubseteq (m^\sharp)_{ij}\big) \\
&\iff \forall i, j.\ \big((m^\sharp)_{ij} = \bigstar \implies (\alpha(M))_{ij} = \bigstar\big) \\
&\iff \forall i, j.\ \big((m^\sharp)_{ij} = \bigstar \implies +\infty \notin \{m_{ij} \mid m \in M \setminus \{\bot\}\}\big) \\
&\iff \forall i, j.\ \big((m^\sharp)_{ij} = \bigstar \implies \forall m \in (M \setminus \{\bot\}).\, m_{ij} \neq +\infty\big) \\
&\iff \forall i, j. \forall m \in (M \setminus \{\bot\}).\ \big((m^\sharp)_{ij} = \bigstar \implies m_{ij} \neq +\infty\big) \\
&\iff \forall m \in (M \setminus \{\bot\}). \forall i, j.\ \big((m^\sharp)_{ij} = \bigstar \implies m_{ij} \neq +\infty\big) \\
&\iff \forall m \in (M \setminus \{\bot\}).\, m \neq \bot \land \forall i, j.\ \big((m^\sharp)_{ij} = \bigstar \implies m_{ij} \neq +\infty\big) \\
&\iff \forall m \in (M \setminus \{\bot\}).\, m \in (\gamma(m^\sharp) \setminus \{\bot\}) \\
&\iff (M \setminus \{\bot\}) \subseteq (\gamma(m^\sharp) \setminus \{\bot\}).
\end{aligned}$$

The pre-analysis uses the abstract semantics $[\![cmd]\!]^\sharp : \mathbb{O}^\sharp \to \mathbb{O}^\sharp$ that is derived from this Galois connection and the abstract semantics of the same command in Octagon (Figure 3). Figure 4 shows the results of this derivation. One non-trivial case is $x_i = x_i + k_{>0}$. The pre-analysis conservatively approximates the upper bound of $x_i - x_j$ to infinity (i.e., $\top$) as x_i increases by $x_i = x_i + k_{>0}$. On the other hand, the upper bound of $x_j - x_i$ is computed as $\bigstar$, because it is always finite in the Octagon analysis. Our choice on the upper bound of $x_i - x_j$ ($\top$ rather than $\bigstar$) is because of a practical issue. Though the pre-analysis is sound with respect to the least fixed point of the Octagon

analysis, a widening operator is used to compute an over-approximation of it in practice. Consider the following code:

```
x = y;
while(*){
  assert(x - y <= 0);
  x = x + 1;
}
```

In this example, the Octagon analysis with the standard widening computes $x - y \leq +\infty$ inside of the loop. To soundly handle such common cases, we defined the semantics for $x_i = x_i + k_{>0}$ approximately to produce $\top$. If we defined the semantics to produce $\bigstar$, the pre-analysis would select the query in the example program. In our experience, this alternative selects too many unprovable queries in practice [16].

Significance of Impact Pre-analysis A good way to see the significance is to recall the use of the so-called collecting semantics in abstract interpretation and to remember how the collecting semantics helps relate the concrete semantics and the abstract semantics of a program analysis. The collecting semantics models the concrete behavior of each program statement as a transformer on *sets* of states. It is used even when all program statements are deterministic and can be interpreted as (partial) functions on states, as commonly done in the work on denotational semantics. This seemingly unnecessary use of the collecting semantics comes from the fact that this semantics lets us formalize the notion of abstraction in terms of subset relation easily, describe the behavior of a program analysis by means of concretization, and relate the concrete semantics and the analysis.

The reason that we use the power-set domain of matrices is nearly identical. Although the Octagon analysis never generates a set of matrices, it can be reinterpreted in a version of collecting semantics where Octagon associates a set of matrices to each program point and its abstract transfer functions map sets of matrices to other sets. Our pre-analysis approximates this collecting semantics of Octagon, and its behavior and approximation can be described using operations on sets of matrices and Galois connection. The goal of the pre-analysis is to compute (a representation of) a set of matrices at each program point that covers the outcome of the Octagon analysis at the point. Thus, a matrix with ∞ should appear in the computed set if the matrix is the result of the Octagon at p. Contrapositively, if our pre-analysis computes the upper bound of $x - y$ as $\bigstar$ at a program point so that no matrices in the computed set contain ∞ in the (x, y) entry, the Octagon analysis infers $x - y \leq n$ for some $n \neq +\infty$. We want to point out that sets of matrices should not be understood in terms of concrete sets of states obtained by the concretization of the Octagon analysis: even when two sets of matrices mean the same set of states via this concretization, they may describe the completely different behaviors of the Octagon analysis.

Automatic Labeling For every $(P_i, Q_i) \in \mathcal{P}$, we run the pre-analysis on P_i, and get an analysis result X_i that maps each program point in P_i to a matrix in $\mathbb{O}^\sharp$. From such X_i, we generate labeled data $\mathcal{D}$ as follows:

$$Q'_i = \{c \mid \exists p.\,(c,p) \in Q_i \wedge p \text{ is a predicate about the upper bound of } x_k - x_j \wedge X_i(c) \neq \bot \wedge X_i(c)_{jk} = \bigstar\},$$

$$\mathcal{D} = \bigcup_{1\le i\le N} \{(P_i, (x_j, x_k), L) \mid L = \oplus \iff \exists c \in Q'_i.\exists l, m \in \{0,1\}.\, X_i(c)_{(2j-l)(2k-m)} = \bigstar\}.$$

Notice that we label (x_j, x_k) with $\oplus$ if tracking their relationship is predicted to be useful for *some* query according to the results of the pre-analysis.

4.2 Features and Classifier

The second step of our method is to represent labeled data $\mathcal{D}$ as a set of boolean vectors marked with $\oplus$ or $\ominus$, and to run an off-the-shelf algorithm for inferring a classifier with this set of labeled vectors. The vector representation assumes a collection of features $\mathbf{f} = \{f_1, \dots, f_m\}$, each of which maps a pair $(P, (x, y))$ of program P and variable pair (x, y) to 0 or 1. The vector representation is the set $\mathcal{V}$ defined as follows:

$$\mathbf{f}(P, (x, y)) = \big(f_1(P, (x, y)), \dots, f_m(P, (x, y))\big) \in \{0,1\}^m,$$
$$\mathcal{V} = \{(\mathbf{f}(P, (x, y)), L) \mid (P, (x, y), L) \in \mathcal{D}\} \in \wp(\{0,1\}^m \times \{\oplus, \ominus\}).$$

An off-the-shelf algorithm computes a binary classifier $\mathcal{C}$ from $\mathcal{V}$:

$$\mathcal{C} : \{0,1\}^m \to \{\oplus, \ominus\}.$$

In our experiments, $\mathcal{V}$ has significantly more vectors marked with $\ominus$ than those marked with $\oplus$. We found that the algorithm for inferring a decision tree [10] worked the best for our $\mathcal{V}$

Table 1 shows the features that we developed for the Octagon analysis and used in our experiments. These features work for real C programs (not just those in the small language that we have used so far in the paper), and they are all symmetric in the sense that $f_i(P, (x, y)) = f_i(P, (y, x))$. Features 1–6 detect good situations where the Octagon analysis *can* track the relationship between variables *precisely*. For example, $f_1(P, (x, y)) = 1$ when x and y appear in an assignment $x = y + k$ or $y = x + k$ for some constant k in the program P. Note that the abstract semantics of these commands in Octagon do not lose any information. The next features 7–11, on the other hand, detect bad situations where the Octagon analysis *cannot* track the relationship between variables *precisely*. For example, $f_7(P, (x, y)) = 1$ when x or y gets multiplied by a constant different from 1 in a command of P, as in the assignments $y = x * 2$ and $x = y * 2$. Notice that these assignments set up relationships between x and y that can be expressed only approximately by Octagon. We have found

i	Description of feature $f_i(P,(x,y))$. k represents a constant.
1	P contains an assignment $x = y + k$ or $y = x + k$.
2	P contains a guard $x \leq y + k$ or $y \leq x + k$.
3	P contains a malloc of the form $x = \texttt{malloc}(y + k)$ or $y = \texttt{malloc}(x + k)$.
4	P contains a command $x = \texttt{strlen}(y)$ or $y = \texttt{strlen}(x)$.
5	P sets x to $\texttt{strlen}(y)$ or y to $\texttt{strlen}(x)$ indirectly, as in $t = \texttt{strlen}(y); x = t$.
6	P contains an expression of the form $x[y + k]$ or $y[x + k]$.
7	P contains an expression that multiplies x or y by a constant different from 1.
8	P contains an expression that multiplies x or y by a variable.
9	P contains an expression that divides x or y by a variable.
10	P contains an expression that has x or y as an operand of bitwise operations.
11	P contains an assignment that updates x or y using non-Octagonal expressions.
12	x and y are has the same name in different scopes.
13	x and y are both global variables in P.
14	x or y is a global variable in P.
15	x or y is a field of a structure in P.
16	x and y represent sizes of some arrays in P.
17	x and y are temporary variables in P.
18	x or y is a local variable of a recursive function in P.
19	x or y is tested for the equality with ± 1 in P.
20	x and y represent sizes of some global arrays in P.
21	x or y stores the result of a library call in P.
22	x and y are local variables of different functions in P.
23	$\{x, y\}$ consists of a local var. and the size of a local array in different fun. in P.
24	$\{x, y\}$ consists of a local var. and a temporary var. in different functions in P.
25	$\{x, y\}$ consists of a global var. and the size of a local array in P.
26	$\{x, y\}$ contains a temporary var. and the size of a local array in P.
27	$\{x, y\}$ consists of local and global variables not accessed by the same fun. in P.
28	x or y is a self-updating global var. in P.
29	The flow-insensitive analysis of P results in a finite interval for x or y.
30	x or y is the size of a constant string in P.

Table 1 Features for relations of two variables.

that detecting both good and bad situations is important for learning an effective variable-clustering strategy. The remaining features (12–30) describe various syntactic and semantics properties about program variables that often appear in typical C programs. For the semantic features, we use the results of a flow-insensitive analysis that quickly computes approximate information about pointer aliasing and ranges of numerical variables.

4.3 Strategy for Clustering Variables

The last step is to define a strategy that takes a program P, especially one not seen during learning, and clusters variables in P. Assume that a program P is given and let Var_P be the set of variables in P. Using features $\mathbf{f}$ and inferred classifier $\mathcal{C}$, our strategy computes the *finest* partition of Var_P,

$$\Pi = \{\pi_1, \ldots, \pi_k\} \subseteq \wp(Var_P),$$

such that for all $(x, y) \in Var_P \times Var_P$, if we let $\mathcal{F} = \mathcal{C} \circ \mathbf{f}$, then

$$\mathcal{F}(P, (x, y)) = \oplus \implies x, y \in \pi_i \text{ for some } \pi_i \in \Pi.$$

The partition Π is the clustering of variables that will be used by the partial Octagon analysis subsequently. Notice that although the classifier does not indicate the importance of tracking the relationship between some variables x and z (i.e., $\mathcal{F}(P,(x,z)) = \ominus$), Π may put x and z in the same $\pi \in \Pi$, if $\mathcal{F}(P,(x,y)) = \mathcal{F}(P,(y,z)) = \oplus$ for some y. Effectively, our construction of Π takes the transitive closure of the raw output of the classifier on variables. In our experiments, taking this transitive closure was crucial for achieving the desired precision of the partial Octagon analysis.

5 Partially Context-sensitive Analysis with Context Selector

In this section, we describe a selective context-sensitive analysis with the interval domain [16]. In this analysis, the degree of context-sensitivity is characterized by a context selector, which will be learned from data in Section 6.

5.1 Programs

We assume that a program is represented by a control-flow graph $(\mathbb{C}, \to, \mathbb{F})$ where $\mathbb{C}$ is the set of program points, $(\to) \subseteq \mathbb{C} \times \mathbb{C}$ denotes the control flows of the program and $\mathbb{F}$ is the set of procedure names. A node $c \in \mathbb{C}$ is associated with one of the following primitive commands:

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

where e is an arithmetic expression:

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

where $n \in \mathbb{Z}$ is an integer and x is a program variable. $\mathbb{C}$ consists of five disjoint sets:

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

Each procedure $f \in \mathbb{F}$ has only one entry and one exit node. Given a node $c \in \mathbb{C}$, $\mathsf{fid}(c)$ denotes the procedure enclosing the node. We denote the set of call nodes by $\mathbb{C}_c \subseteq \mathbb{C}$ and the set of program variables by Var.

5.2 Interval Analysis

The abstract domain $\mathbb{S}$ for the interval analysis is defined as follows:

$$\mathbb{S} = Var \rightarrow \mathbb{I}$$

Abstract domain $\mathbb{I}$ is the standard interval domain. The domain for abstract states is a map from variables to intervals.

The abstract semantics of primitive commands $[\![cmd]\!] : \mathbb{S} \rightarrow \mathbb{S}$ is defined as follows:

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

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

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

5.3 Context Selector and Partial Context-sensitivity

A context selector K is a set of procedures that are treated context-sensitively by the analysis:

$$K \in \wp(\mathbb{F})$$

We extend the control-flow graph with context selector K. First, we define a set $\mathbb{C}_K \subseteq \mathbb{C} \times \mathbb{C}_c^*$ of context-enriched nodes. With the definition of $\mathbb{C}_K$, the control flow relation $(\rightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ is extended to $\rightarrow_K$:

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

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

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

where ϵ is the empty call sequence that subsumes all other contexts.

The analysis differentiates multiple abstract states at each program node c, depending on each context κ. The analysis computes a pre-fixpoint of the following abstract semantic function F on $\mathbb{C}_K$:

$$F : (\mathbb{C}_K \rightarrow \mathbb{S}) \rightarrow (\mathbb{C}_K \rightarrow \mathbb{S})$$

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

6 Learning a Strategy for Selecting Contexts

In this section, we present our method for learning a context selector from a given codebase. As in the case of Section 4, we assume that a set of pairs of programs and queries:

$$\mathcal{P} = \{(P_1, Q_1), \ldots, (P_N, Q_N)\},$$

A query in Q_i is a pair of a program point c of P_i and a predicate p on a variable in P_i. In our setting, the predicate indicates whether the value of the variable is positive or not, i.e., $x \geq 1$. The choice is because our client analysis targets to buffer-overrun properties that requires the size of buffer should be a positive number. The general principle is also applicable to other client analyses by using different predicates.

6.1 Automatic Generation of Labeled Data

Impact Pre-analysis We review the impact pre-analysis for context-sensitivity [16]. The impact pre-analysis estimates the impact of context-sensitivity of the main analysis. To do so, the pre-analysis is fully context-sensitive but uses a simple finite abstract domain:

$$\mathbb{S}^\sharp = \mathit{Var} \rightarrow \mathbb{V}$$

where $\mathbb{V} = \{\bot, \bigstar, \top\}$ is a finite complete lattice. The meaning of $\mathbb{V}$ is characterized by the following function:

$$\begin{aligned} \gamma_v(\top) &= \mathbb{I} \\ \gamma_v(\bigstar) &= \{[a, b] \in \mathbb{I} \mid 0 \leq a\} \\ \gamma_v(\bot) &= \emptyset \end{aligned}$$

Domain $\mathbb{V}$ abstracts a set of intervals. $\bigstar$ denotes all non-negative intervals and $\top$ denotes all intervals.

The abstract semantics for the impact pre-analysis is defined as follows:

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

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

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

The analysis computes the least fixpoint of the semantic function F:

$$F^\sharp : (\mathbb{C}_{K_\infty} \to \mathbb{S}^\sharp) \to (\mathbb{C}_{K_\infty} \to \mathbb{S}^\sharp)$$
$$F^\sharp(X)(c,\kappa) = [\![cmd(c)]\!]^\sharp(\bigsqcup_{(c_0,\kappa_0)\to_{K_\infty}(c,\kappa)} X(c_0,\kappa_0)).$$

where $K_\infty = \mathbb{F}$ represents the full context-sensitivity.

Automatic Labeling We run the pre-analysis on every program in the codebase and collect all the functions that are predicted to be effective for proving some queries. To do so, we first select queries according to the pre-analysis results:

$$Q'_i = \{(c,x) \mid (c,x) \in Q \ \wedge \ X_i(c)(x) = \bigstar\}$$

where X_i is a pre-analysis result that maps each program point in P_i to an abstract state in $\mathbb{S}^\sharp$. Next, we build a program slice that includes all the dependencies of each selected query. All the functions that are involved in the dependencies are marked as positive examples.

The slice of a query is defined on the value-flow graph $(\Theta, \hookrightarrow)$ [16] of the given program as follows:

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

A node of the value-flow graph is a pair of program node and a variable, and $(\hookrightarrow)$ is the edge relation between the nodes.

Definition 2 ($\hookrightarrow$) The value-flow relation $(\hookrightarrow) \subseteq (\mathbb{C} \times \mathit{Var}) \times (\mathbb{C} \times \mathit{Var})$ links the vertices in $\mathbb{C} \times \mathit{Var}$ according to the specification below:

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

where $\mathsf{var}(e)$ is defined as follows:

$$\mathsf{var}(e) = \{x_1, \cdots, x_n\} \iff \forall s \in \mathbb{S}^\sharp \setminus \{\bot\}. \exists v \in \mathbb{V}. [\![e]\!]^\sharp(s) = s(x_1) \sqcup \cdots s(x_n) \sqcup v$$

We extend the $\hookrightarrow$ to its (fully) context-enriched version $\hookrightarrow_K$:

Definition 3 ($\hookrightarrow_K$) The (fully) context-enriched value-flow relation $(\hookrightarrow_{K_\infty}) \subseteq (\mathbb{C}_{K_\infty} \times \mathit{Var}) \times (\mathbb{C}_{K_\infty} \times \mathit{Var})$ links the vertices in $\mathbb{C}_{K_\infty} \times \mathit{Var}$ according to the specification below:

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

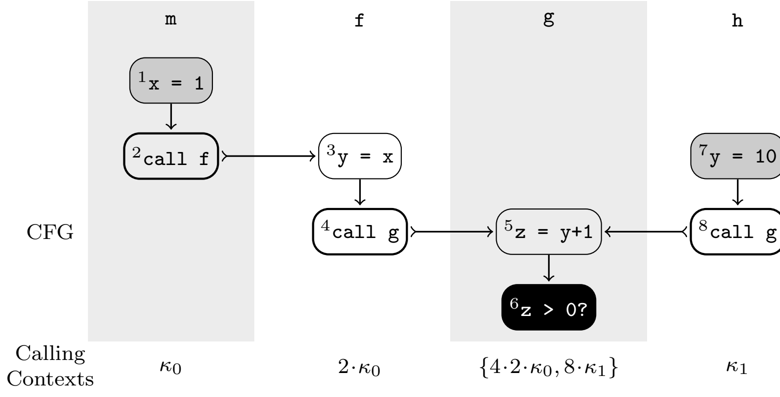


Fig. 5 Example context selector. Gray and black nodes in CFG are source and query points, respectively. The superscript in front of each command denotes the control point.

On the value-flow graph, we derive 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:

$$\exists \kappa, \kappa_q. \ (\iota, \epsilon) \rightarrow_K^* (c, \kappa) \ \wedge \ ((c, \kappa), x) \hookrightarrow_K^* ((c_q, \kappa_q), x_q).$$

Among such dependent nodes, we denote the set of nodes that do not have predecessors by *sources* (Φ):

Definition 4 **(Φ)** Sources Φ are vertices in Θ where dependencies begin:

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

Intuitively the sources implies that the abstract semantics at $(c_0, x_0) \in \Theta$ assigns a fixed constant abstract value to x_0 without using or joining other abstract values from vertices in Θ.

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

Definition 5 **$(\Phi_{(c_q,x_q)})$** Sources on which the query (c_q, x_q) depends:

$$\Phi_{(c_q,x_q)} = \{(c_0, x_0) \in \Phi \mid \exists \kappa_0, \kappa_q. (\iota, \epsilon) \rightarrow_K^* (c_0, \kappa_0) \wedge (c_0, x_0) \hookrightarrow_K^* ((c_q, \kappa_q), x_q)\}.$$

Example 1 Consider the control flow graph in Figure 5. Node 6 denotes the query point, i.e., $(c_q, x_q) = (6, \mathtt{z})$. The gray nodes (node 1 and 7) represent the sources of the query, 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

$$((c_0, \kappa_0), x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q, \kappa_q), x_q) \tag{4}$$

represents a dependency path for the query (c_q, x_q).

We denote the set of all dependency paths for the query by $\mathsf{Path}_{(c_q,x_q)}$:

Definition 6 ($\mathsf{Path}_{(c_q,x_q)}$) The set of all dependency paths for the query (c_q, x_q) is defined as follows:

$$\mathsf{Path}_{(c_q,x_q)} = \{((c_0,\kappa_0),x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q,\kappa_q),x_q) \mid (c_0,x_0) \in \Phi_{(c_q,x_q)}\}.$$

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

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

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

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

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

Finally, we compute a set of functions that constitute the context-sensitivity for the query q:

Definition 7 (Fun_q) The set of functions that are called in all the dependency paths for the query q is defined as follows:

$$\mathsf{Fun}_q = \{\mathsf{fid}(c') \mid c \to c' \ \wedge\ c \in \mathbb{C}_c \ \wedge\ c' \in \mathbb{C}_e \ \wedge\ ((_, c\cdot_),_) \in p \ \wedge\ p \in \mathsf{Path}_q\}$$

Example 3 In Figure 5, we compute a set of functions that are called in path p_1 and p_2 in Example 2:

$$\mathsf{Fun}_{(6,z)} = \{f, g\}$$

From the pre-analysis results and the dependencies, we generate labeled data $\mathcal{D}$ as follows:

$$\mathcal{D} = \bigcup_{1\le i\le N} \{(P_i,\psi,L) \mid L = \oplus \iff \exists q \in Q'_i.\ \psi \in \mathsf{Fun}_q\}$$

6.2 Features and Classifier

As in Section 4.2, we represent the labeled data as feature vectors and derive a classifier using an off-the-shelf learning algorithm. For context-sensitivity, we use the features for functions in Table 2 [17]. Each of them maps a pair (P, ψ) of program P and function ψ to 0 or 1. The vector representation is the set $\mathcal{V}$ defined as follows:

$$\begin{aligned} &\mathbf{f}(P,\psi) = \big(f_1(P,\psi),\ldots,f_m(P,\psi)\big) \in \{0,1\}^m, \\ &\mathcal{V} = \{(\mathbf{f}(P,\psi),L) \mid (P,\psi,L) \in \mathcal{D}\} \in \wp(\{0,1\}^m \times \{\oplus,\ominus\}). \end{aligned}$$

An off-the-shelf algorithm computes a binary classifier $\mathcal{C}$ from $\mathcal{V}$:

$$\mathcal{C} : \{0,1\}^m \to \{\oplus,\ominus\}.$$

i	Description of feature $f_i(P, \psi)$
1	leaf function
2	function containing malloc
3	function containing realloc
4	function containing a loop
5	function containing an if statement
6	function containing a switch statement
7	function using a string-related library function
8	write to a global variable
9	read a global variable
10	write to a structure field
11	read from a structure field
12	directly return a constant expression
13	indirectly return a constant expression
14	directly return an allocated memory
15	indirectly return an allocated memory
16	directly return a reallocated memory
17	indirectly return a reallocated memory
18	return expression involves field access
19	return value depends on a structure field
20	return void
21	directly invoked with a constant
22	constant is passed to an argument
23	invoked with an unknown value
24	functions having no arguments
25	functions having one argument
26	functions having more than one argument
27	functions having an integer argument
28	functions having a pointer argument
29	functions having a structure as an argument

Table 2 Features for partially context-sensitive analysis.

6.3 Strategy for Context Selector

Finally, we define a strategy for context-sensitivity. Given a program P and a set of functions in P, which is denoted by $\mathbb{F}_P$, our strategy computes the following set of functions as the context selector K:

$$K = \{\psi \mid \mathcal{C} \circ \mathbf{f}(\psi) = \oplus \;\wedge\; \psi \in \mathbb{F}_P\}$$

7 Experiments

We describe the experimental evaluation of our method for learning strategies for variable-clustering and context-sensitivity. The evaluation aimed to answer the following questions:

1. **Effectiveness**: How well does the partially relational Octagon and partially context-sensitive Interval analysis with a learned strategy perform, compared with the existing Interval and Octagon analyses?
2. **Generalization**: Does the strategy learned from small programs also work well for large unseen programs?

3. **Feature design**: How should we choose a set of features in order to make our method learn a good strategy?
4. **Choice of an off-the-shelf classification algorithm**: Our method uses a classification algorithm for inferring a decision tree by default. How much does this choice matter for the performance of our method?

We conducted our experiments with a realistic static analyzer and open-source C benchmarks. We implemented our method on top of Sparrow, a static buffer-overflow analyzer for real-world C programs [27]. The analyzer performs the combination of the Interval analysis and the pointer analysis based on allocation-site abstraction with several precision-improving techniques such as fully flow-, field-sensitivity and selective context-sensitivity [16]. In our experiments, we modified Sparrow to use the partially relational Octagon analysis as presented in Section 3 and the partially context-sensitive Interval analysis as presented in Section 5. The analyses were implemented on top of the sparse analysis framework [15,14], so it is significantly faster than the vanilla analyses. For the implementation of data structures and abstract operations for Octagon, we tried the OptOctagons plugin [26] of the Apron framework [7]. For the decision tree learning, we used the Scikit-learn [18] library and its default hyperparameters.

We used 25 open-source benchmark programs (Table 3). Among the benchmark programs, we selected two subsets of the programs that may benefit from context-sensitivity and Octagon, respectively. To do so, we ran their impact pre-analysis counterparts [16], then chose 22 programs for the partially context-sensitive analysis and 17 programs for the partially relational Octagon analysis according to the pre-analysis results. All the experiments were done on a Ubuntu machine with Intel Xeon clocked at 2.4GHz cpu and 192GB of main memory.

7.1 Effectiveness

We evaluated the effectiveness of a strategy learned by our method on the cost and precision of the analyses.

7.1.1 Partially Relational Octagon Analysis

We compared the partial Octagon analysis with a learned variable-clustering strategy with the Interval analysis and the approach for optimizing Octagon in [16]. The approach in [16] also performs the partial Octagon analysis in Section 3 but with a fixed variable-clustering strategy that uses the impact pre-analysis online (rather than offline as in our case): the strategy runs the impact pre-analysis on a given program and computes variable clusters of the program based on the results of the pre-analysis. Note that this partial Octagon analysis only based on the pre-analysis is already significantly faster than the original Octagon analysis and the partial Octagon analysis based on a syntactic heuristic [1]. All the three analysis are partially context-sensitive enabled by

Program	LOC	#Var	#Fun
brutefir-1.0f	103	54	2
consolcalculator-1.0	298	165	5
id3-0.15	512	527	2
spell-1.0	2,284	427	30
mp3rename-0.6	2,466	332	7
irmp3-0.5.3.1	3,797	523	82
barcode-0.96	4,460	1,738	57
httptunnel-3.3	6,174	1,622	80
e2ps-4.34	6,222	1,437	15
sbm-0.0.4	6,502	868	64
mpegdemux-0.1.3	7,783	732	71
bzip2-spec2000	9,796	945	75
bc-1.06	13,093	1,891	133
less-382	23,822	3,682	382
tar-1.13	30,154	4,944	222
agedu-8642	32,637	4,149	86
gbsplay-0.0.91	34,002	1,608	188
bison-2.5	56,361	14,610	964
pies-1.2	66,196	9,472	701
icecast-server-1.3.12	68,564	6,183	562
aalib-1.4p5	73,412	2,786	129
raptor-1.4.21	76,378	8,889	976
dico-2.0	84,333	4,349	622
rnv-1.7.10	93,858	4,146	390
lsh-2.0.4	110,898	18,880	1,702

Table 3 The characteristics of the benchmark programs. **LOC** reports lines of code before preprocessing. **#Var** reports the number of program variables (more precisely, abstract locations). **#Fun** reports the number of functions in each program.

the pre-analysis for context-sensitivity [16]. To show the net effects of variable clustering, we omitted the cost of the pointer analysis and compared only the cost for the numerical analyses. Table 4 shows the results of our comparison with 17 open-source programs. We used the leave-one-out cross validation to evaluate our method; for each program P in the table, we applied our method to the other 16 programs, learned a variable-clustering strategy, and ran the partial Octagon on P with this strategy.

The results show that the partial Octagon with a learned strategy strikes the right balance between precision and cost. In total, the Interval analysis reports 7,406 alarms from the benchmark set.[3] The existing approach for partial Octagon [16] reduced the number of alarms by 252, but increased the analysis time by 62x. Meanwhile, our learning-based approach for partial Octagon reduced the number of alarms by 240 while increasing the analysis time by 2x.

We point out that in some programs, the precision of our approach was incomparable with that of the approach in [16]. For instance, our approach would produce less precise results if the usage patterns of variables did not appear in training programs. The following code is excerpted from `less-382`:

[3] In practice, eliminating these false alarms is extremely challenging in a sound yet non-domain-specific static analyzer for full C. The false alarms arise from a variety of reasons, e.g., recursive calls, unknown library calls, complex loops, etc.

Program	#Alarms			Time(s)		
	Itv	Impt	ML	Itv	Impt	ML
brutefir-1.0f	4	0	0	0	0 (0)	0 (0)
consolcalculator-1.0	20	10	10	0	0 (0)	0 (0)
id3-0.15	15	6	6	0	0 (0)	1 (0)
spell-1.0	20	8	17	0	1 (1)	1 (0)
mp3rename-0.6	33	3	3	0	1 (0)	1 (0)
irmp3-0.5.3.1	2	0	0	1	2 (0)	3 (1)
barcode-0.96	235	215	215	2	9 (7)	6 (1)
httptunnel-3.3	52	29	27	3	35 (32)	5 (1)
e2ps-4.34	119	58	58	3	6 (3)	3 (0)
bc-1.06	371	364	364	14	252 (238)	16 (1)
less-382	625	620	625	83	2,354 (2,271)	87 (4)
bison-2.5	1,988	1,955	1,955	137	4,827 (4,685)	237 (79)
pies-1.2	795	785	785	49	14,942 (14,891)	95 (43)
icecast-server-1.3.12	239	232	232	51	109 (55)	107 (42)
raptor-1.4.21	2,156	2,148	2,148	242	17,844 (17,604)	345 (104)
dico-2.0	402	396	396	38	156 (117)	51 (24)
lsh-2.0.4	330	325	325	33	139 (106)	251 (218)
Total	7,406	7,154	7,166	656	40,677 (40,011)	1,207 (519)

Table 4 Comparison of performance of the Interval analysis and two partial Octagon analyses, one with a fixed strategy based on the impact pre-analysis and the other with a learned strategy. **#Alarms** reports the number of buffer-overflow alarms reported by the interval analysis (**Itv**), the partial Octagon analysis with a fixed strategy (**Impt**) and that with a learned strategy (**ML**). **Time** shows the analysis time in seconds, where, in **X**(**Y**), **X** means the total time (including that for clustering and the time for main analysis) and **Y** shows the time spent by the strategy for clustering variables.

```
1 int old_lesskey(char* buf, int len){
2   *(buf + (len - 1)) = 0;  // Query
3 }
4
5 int lesskey(){
6   buf = malloc(len);
7   old_lesskey(buf, len);
8 }
```

The pre-analysis precisely estimated that the query at line 2 can be proven by the Octagon analysis. However, our learning approach could not infer this fact because in training programs, it was less common 1) to pass a buffer and its size to a callee and then 2) to access an element inside of the callee. On the other hand, for `httptunnel-3.3`, our approach produces better results because the impact pre-analysis of [16] uses ★ conservatively and fails to identify some important relationships between variables as in Section 4. Consider the following code:

```
1 data = malloc(n + 2);
2 n = n + 1;
3 data[n] = 0;    // Query
```

The pre-analysis over-approximates the upper-bound of the difference between `n` and the size of `data` to $\top$ because of the assignment at line 2. As a result,

the approach in [16] fails to find out that the Octagon analysis is able to prove the safety of the memory access at line 3. However, our approach can prove the query because it works by identifying relationships between variables to be tracked by Octagon, instead of finding out which queries would be proved by Octagon. When our approach summarizes this example using features, it notices that an assignment of the form $x = y + k$, a malloc of the form $x = \texttt{malloc}(y + k)$, and an expression of the form $x[y + k]$ are used in the example. Then, using the learned classifier, it infers that the relationship between `data` and `n` should be tracked because these two variables are involved in those assignment, malloc and array access. This inference makes the following Octagon analysis prove the query.

7.1.2 Partially Context-sensitive Interval Analysis

We evaluated the performance of the partially context-sensitive Interval analysis with our learned strategy compared to the context-insensitive one and the partially context-sensitive one by the impact pre-analysis. Like the Octagon experiments in the previous section, we used the leave-one-out cross validation on 22 programs to evaluate the learning method.

We came to the same conclusion; our learning-based strategy shows similar precision but noticeably saved the analysis time. Table 5 shows the partially context-sensitive analysis with a learned strategy is cost-effective. In total, the context-insensitive Interval analysis reported 13,663 alarms form the benchmark set. The partially context-sensitive one by the pre-analysis [16] reduced 3,387 alarms (24.8%) but increased the analysis time by 40%. Meanwhile, our learning-based approach reduced the number of alarms by 3,382 (24.8%) while increasing the time consumption by only 14.1%.

7.2 Generalization

Although the impact pre-analyses scales far better than the fully sensitive ones, sometime they are still too expensive to be used for training routinely for large programs (> 100 KLOC), (e.g., Octagon). Therefore, in order for our approach to scale, the strategies learned from a codebase of small programs need to be effective for large unseen programs. Whether this need is met or not depends on whether our learning method generalize information from small programs to large programs well.

To evaluate this generalization capability of our learning method, we divided the benchmark set into small (< 60 KLOC) and large (> 60 KLOC) programs, learned each strategy from the group of small programs, and evaluated its performance on that of large programs.

Table 6 shows the results of the partial Octagon analysis. Columns labeled **Small** report the performance of our approach learned from the small programs. **All** reports the performance of the strategy used in Section 7.1 (i.e., the strategy learned with all benchmark programs except for each target

Program	#Alarms			Time(s)		
	CI	Impt	ML	Itv	Impt	ML
spell-1.0	58	20	20	1	11 (0)	2 (0)
mp3rename-0.6	35	33	33	1	2 (1)	1 (0)
irmp3-0.5.3.1	2	2	2	1	4 (2)	3 (1)
barcode-0.96	235	235	235	2	6 (2)	4 (1)
httptunnel-3.3	51	51	51	7	9 (2)	7 (1)
e2ps-4.34	119	119	119	3	8 (2)	3 (0)
sbm-0.0.4	174	174	174	1	12 (2)	2 (0)
mpegdemux-0.1.3	61	61	61	2	13 (1)	2 (0)
bzip2-spec2000	270	266	268	4	4 (1)	4 (1)
bc-1.06	614	371	371	18	32 (8)	49 (2)
less-382	625	625	625	390	372 (13)	359 (10)
tar-1.13	932	683	683	128	130 (14)	103 (3)
agedu-8642	688	467	467	16	129 (115)	17 (3)
gbsplay-0.0.91	74	74	72	5	5 (1)	5 (1)
bison-2.5	3,497	1,988	1,989	335	448 (80)	398 (30)
pies-1.2	1,508	795	794	128	175 (24)	142 (10)
icecast-server-1.3.12	288	239	239	98	194 (48)	136 (8)
aalib-1.4p5	126	126	126	8	18 (9)	24 (16)
raptor-1.4.21	2,322	2,156	2,160	606	753 (92)	751 (54)
dico-2.0	584	402	403	37	43 (2)	54 (11)
rnv-1.7.10	1,067	1,059	1,062	35	66 (18)	53 (6)
lsh-2.0.4	333	330	327	125	299 (174)	151 (20)
Total	13,663	10,276	10,281	1,951	2,733 (611)	2,270 (178)

Table 5 Comarison of performance of the context-insensitive Interval analysis and two partially context-sensitive ones, one with a fixed strategy based on the impact pre-analysis and the other with a learned strategy. **#Alarms** reports the number of buffer-overflow alarms reported by the context-insensitive interval analysis (**CI**), the partial context-sensitive analysis with a fixed strategy (**Impt**) and that with a learned strategy (**ML**). **Time** shows the analysis time in seconds, where, in **X**(**Y**), **X** means the total time (including that for the selection and the time for main analysis) and **Y** shows the time spent by the selection.

Program	#Alarms			Time(s)		
	Itv	All	Small	Itv	All	Small
pies-1.2	795	785	785	49	95 (43)	98 (43)
icecast-server-1.3.12	239	232	232	51	113 (42)	99 (42)
raptor-1.4.21	2,156	2,148	2,148	242	345 (104)	388 (104)
dico-2.0	402	396	396	38	61 (24)	62 (24)
lsh-2.0.4	330	325	325	33	251 (218)	251 (218)
Total	3,922	3,886	3,886	413	864 (432)	899 (432)

Table 6 Generalization performance of our learning method for the partial Octagon analysis. **Small** reports the performance of our technique using only small programs ($<$ 60KLOC) as training data. Other results by the interval analysis (**Itv**) and the leave-one-out cross validation (**All**) are the same as Table 4.

program). In our experiments, **Small** had the same precision as **All** with negligibly increase in analysis time (4%). These results show that the information learned from small programs is general enough to infer the useful properties about large programs.

Table 7 shows the results of the partially context-sensitive Interval analysis. In the experiments, **Small** did not seriously sacrifice the performance: it had

	#Alarms			Time(s)		
Program	**CI**	**All**	**Small**	**CI**	**All**	**Small**
pies-1.2	1,508	795	791	128	142 (10)	157 (10)
icecast-server-1.3.12	288	239	239	98	136 (8)	122 (8)
aalib-1.4p5	126	126	125	8	24 (16)	40 (16)
raptor-1.4.21	2,322	2,156	2,160	606	751 (54)	891 (54)
dico-2.0	584	402	402	37	54 (11)	52 (11)
rnv-1.7.10	1,067	1,059	1,062	35	53 (6)	73 (6)
lsh-2.0.4	333	330	333	125	151 (20)	159 (20)
Total	6,228	5,107	5112	1,037	1,311 (125)	1494 (125)

Table 7 Generalization performance of our learning method for the partially context-sensitive Interval analysis. **Small** reports the performance of our technique using only small programs (< 60KLOC) as training data. Other results by the context-sensitive interval analysis (**CI**) and the leave-one-out cross validation (**All**) are the same as Table 5.

almost the same precision as **All** while increasing in analysis time by 13%. In this case, the performance degradation is larger than the partial Octagon case. We conjecture that this is because of the quantity and quality of the training data. The number of training data of partial context-sensitivity (functions) is much less than the ones for partial Octagon (variable pairs). In addition, the pre-analysis for context-sensitivity is less precise than the one for Octagon [16].

7.3 Feature Design

We identified top ten features that are most important to learn an effective variable-clustering strategy and an context-sensitivity. We applied our method to all the programs so as to learn a decision tree, and measured the relative importance of features by computing their Gini index [2] with the tree. Intuitively, the Gini index shows how much each feature helps a learned decision tree to classify variable pairs as $\oplus$ or $\ominus$. Thus, features with high Gini index are located in the upper part of the tree.

According to the results, the ten most important features are 30, 15, 18, 16, 29, 6, 24, 23, 1, and 21 in Table 1. We found that many of the top ten features are negative and describe situations where the precise tracking of variable relationships by Octagon is unnecessary. For instance, feature 30 (size of constant string) and 29 (finite interval) represent variable pairs whose relationships can be precisely captured even with the Interval analysis. Using Octagon for such pairs is overkill. Initially, we conjectured that positive features, which describe situations where the Octagon analysis is effective, would be the most important for learning a good strategy. However, data show that effectively ruling out unnecessary variable relationships is the key to learning a good variable-clustering strategy for Octagon.

In context-sensitivity, the top features are 2, 3, 8, 14, 15, 16, 17 21, 5, and 9 in Table 2. Most of the top features describe common scenarios that context-sensitivity helps. For example, the following code in Figure 6 has a wrapper of `xmalloc` and an identity function `dec` that require context-sensitivity for

```
1  char* xmalloc(int size){
2    p = malloc(size);
3    if(p == 0) fail();
4    else return p;
5  }
6
7  char* dec(int x){ return x-1; }
8
9  void main(){
10   p = xmalloc(4);
11   q = xmalloc(unknown);
12   i = id(4);
13   j = id(unknown);
14   *(p+i) = 0;
15   *(q+j) = 0;
16 }
```

Fig. 6 A common pattern that context-sensitivity helps.

precise analysis. To prove the safety of the buffer access at line 14, the analysis runs with context-sensitivity on `xmalloc` and `dec` that have feature 2 (contain `malloc`), 14 (return an allocated memory), 21 (invoke with a constant), and 5 (contain an if statement).

7.4 Choice of an off-the-shelf classification algorithm

Our learning method uses an off-the-shelf algorithm for inferring a decision tree. In order to see the importance of this default choice, we replaced the decision-tree algorithm by logistic regression [11], which is another popular supervised learning algorithm and infers a linear classifier from labeled data. Such linear classifiers are usually far simpler than nonlinear ones such as a decision tree. We then repeated the leave-one-out cross validation described in Section 7.1.

In this experiment, the new partial Octagon analysis with linear classifiers proved the same number of queries as before, but it was significantly slower than the analysis with decision trees. Changing regularization in logistic regression from nothing to L_1 or L_2 and varying regularization strengths (10^{-3}, 10^{-4} and 10^{-5}) did not remove this slowdown. We observed that in all of these cases, inferred linear classifiers labeled too many variable pairs with $\oplus$ and led to unnecessarily big clusters of variables. Such big clusters increased the analysis time of the partial Octagon with decision trees by 10x–12x. Such an observation indicates that a linear classifier is not expressive enough to identify important variable pairs for the Octagon analysis.

8 Related Work

8.1 Octagon analysis

The scalability issue of the Octagon analysis is well-known, and there have been various attempts to optimize the analysis [15,26]. Oh et al. [15] exploited the data dependencies of a program and removed unnecessary propagation of information between program points during Octagon's fixpoint computation. Singh et al. [26] designed better algorithms for Octagon's core operators and implemented a new library for Octagon called OptOctagons, which has been incorporated in the Apron framework [7]. These approaches are orthogonal to our approach, and all of these three can be used together as in our implementation. We point out that although the techniques from these approaches [15, 26] improve the performance of Octagon significantly, without additionally making Octagon partial with good variable clusters, they were not enough to make Octagon scale large programs in our experiments. This is understandable because the techniques keep the precision of the original Octagon while making Octagon partial does not.

Existing variable-clustering strategies for the Octagon analysis use a simple syntactic criterion for clustering variables [1] (such as selecting variable pairs that appear in particular kinds of commands and forming one cluster for each syntactic block), or a pre-analysis that attempts to identify important variable pairs for Octagon [16]. When applied to large general-purpose programs (not designed for embedded systems), the syntactic criterion led to ineffective variable clusters, which made the subsequent partial Octagon analysis slow and fail to achieve the desired precision [16]. The approach based on the pre-analysis [16], on the other hand, has an issue with the cost of the pre-analysis itself; it is cheaper than that of Octagon, but it is still expensive as we showed in the paper. In a sense, our approach automatically learns fast approximation of the pre-analysis from the results of running the pre-analysis on programs in a given codebase. In our experiments, this approximation (which we called strategy) was 33x faster than the pre-analysis while decreasing the number of proved queries by 2% only.

8.2 Data-driven approach to program analysis

Recently there have been a large amount of research activities for developing data-driven approaches to challenging program analysis problems, such as specification inference [21,19], invariant generation [12,20,23–25,4], acceleration of abstraction refinement [5], and smart report of analysis results [8,13, 29]. In particular, Oh et al. [17] considered the problem of automatically learning analysis parameters from a codebase, which determine the heuristics used by the analysis. They formulated this parameter learning as a blackbox optimization problem, and proposed to use Bayesian optimization for solving the problem. Initially we followed this blackbox approach [17], and tried Bayesian

optimization to learn a good variable-clustering strategy with our features. In the experiment, we learned the strategy from the small programs as in Section 7.2 and chose the top 200 variable pairs which are enough to make a good clustering as precise as our strategy; the learning process was too costly with larger training programs and more variable pairs. This initial attempt was a total failure. The learning process tried only 384 parameters and reduced 14 false alarms even during the learning phase for a whole week, while our strategy reduced 240 false alarms. Unlike the optimization problems for the analyses in [17], our problem was too difficult for Bayesian optimization to solve. We conjecture that this was due to the lack of smoothness in the objective function of our problem. This failure led to the approach in this paper, where we replaced the blackbox optimization by a much easier supervised-learning problem.

9 Conclusion

In this paper we proposed a method for learning a variable-clustering strategy for the Octagon analysis from a codebase. One notable aspect of our method is that it generates labeled data automatically from a given codebase by running the impact pre-analysis for Octagon [16]. The labeled data are then fed to an off-the-shelf classification algorithm (in particular, decision-tree inference in our implementation), which infers a classifier that can identify important variable pairs from a new unseen program, whose relationships should be tracked by the Octagon analysis. This classifier forms the core of the strategy that is returned by our learning method. Our experiments show that the partial Octagon analysis with the learned strategy scales up to 100KLOC and is 33x faster than the one with the impact pre-analysis (which itself is significantly faster than the original Octagon analysis), while increasing false alarms by only 2%. The general principle of this method is applicable to other types of sensitivities in static analysis. We experimentally demonstrate that the method is also effective for a context-sensitive interval analysis.

We also show that the general idea of this approach is applicable to learn a strategy for other types of sensitivities in static analysis. We designed a partially context-sensitive interval analysis using our learning method with the same idea. The experimental results show that our method is also effective to learn a strategy for context-sensitivity.

Acknowledgements This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Numbers SRFC-IT1701-09. This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00565,Development of Vulnerability Discovery Technologies for IoT Software Security, No.2017-0-00184, Self-Learning Cyber Immune Technology Development).

References

1. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A Static Analyzer for Large Safety-Critical Software. In: PLDI (2003)
2. Breiman, L.: Random Forests. Machine Learning (2001)
3. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Restraints among Variables of a Program. In: POPL (1978)
4. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: POPL, pp. 499–512 (2016)
5. Grigore, R., Yang, H.: Abstraction refinement guided by a learnt probabilistic model. In: POPL (2016)
6. Heo, K., Oh, H., Yang, H.: Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In: SAS (2016)
7. Jeannet, B., Miné, A.: Apron: A Library of Numerical Abstract Domains for Static Analysis. In: CAV (2009)
8. Mangal, R., Zhang, X., Nori, A.V., Naik, M.: A user-guided approach to program analysis. In: ESEC/FSE, pp. 462–473 (2015)
9. Miné, A.: The octagon abstract domain. Higher-order and symbolic computation (2006)
10. Mitchell, T.M.: Machine learning. McGraw-Hill, Inc. (1997)
11. Murphy, K.P.: Machine learning: a probabilistic perspective (adaptive computation and machine learning series). Mit Press ISBN (2012)
12. Nori, A.V., Sharma, R.: Termination proofs from tests. In: FSE, pp. 246–256 (2013)
13. Octeau, D., Jha, S., Dering, M., McDaniel, P., Bartel, A., Li, L., Klein, J., Le Traon, Y.: Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In: POPL, pp. 469–484 (2016)
14. Oh, H., Heo, K., Lee, W., Lee, W., Park, D., Kang, J., Yi, K.: Global sparse analysis framework. ACM Trans. Program. Lang. Syst. **36**(3), 8:1–8:44 (2014). DOI 10.1145/2590811. URL `http://doi.acm.org/10.1145/2590811`
15. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for C-like languages. In: PLDI (2012)
16. Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. In: PLDI (2014)
17. Oh, H., Yang, H., Yi, K.: Learning a strategy for adapting a program analysis via bayesian optimisation. In: OOPSLA (2015)
18. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, É.: Scikit-learn: Machine Learning in Python. The Journal of Machine Learning Research (2011)
19. Raychev, V., Bielik, P., Vechev, M.T., Krause, A.: Learning programs from noisy data. In: POPL, pp. 761–774 (2016)
20. Sankaranarayanan, S., Chaudhuri, S., Ivančić, F., Gupta, A.: Dynamic inference of likely data preconditions over predicates by tree learning. In: ISSTA, pp. 295–306 (2008)
21. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Mining library specifications using inductive logic programming. In: ICSE, pp. 131–140 (2008)
22. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–234. Prentice-Hall, Englewood Cliffs, NJ (1981)
23. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: ESOP, pp. 574–592 (2013). URL `http://dx.doi.org/10.1007/978-3-642-37036-6_31`
24. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: SAS, pp. 388–411 (2013)
25. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: CAV, pp. 71–87 (2012)
26. Singh, G., Püschel, M., Vechev, M.: Making Numerical Program Analysis Fast. In: PLDI (2015)
27. Sparrow: `http://ropas.snu.ac.kr/sparrow`
28. Venet, A., Brat, G.: Precise and efficient static array bound checking for large embedded C programs. In: PLDI (2004)

29. Yi, K., Choi, H., Kim, J., Kim, Y.: An empirical study on classification methods for alarms from a bug-finding static C analyzer. Information Processing Letters **102**(2-3), 118–123 (2007)