Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis

Kihong Heo[1], Hakjoo Oh[2], and Hongseok Yang[3]

[1] Seoul National University
[2] Korea University
[3] University of Oxford

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%.

1 Introduction

Relational program analyses track sophisticated relationships among program variables and enable the automatic verification of complex properties of programs [3, 8]. 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 [8] 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 [8, 15, 26, 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 [26, 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 [15].

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 [15], 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 [9], 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 gets 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 [15] (which was already significantly faster than the original Octagon analysis). The price of speed-up was mere 2% increase of false alarms.

We summarize the contributions of this paper below:

1. We propose a method for automatically learning an effective variable-clustering strategy 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 [15]. 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 [15] while increasing false alarms by only 2%.

2 Informal Explanation

2.1 Octagon Analysis with Variable Clustering

We start with informal explanation of our approach 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.

The Octagon analysis [8] 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:

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

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

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

Fig. 1. Example Program

the matrix respectively and they are variables with or without the minus sign. The matrix records -1 and ∞ as upper bounds for $\mathtt{i} - \mathtt{a}$ and $\mathtt{i} - \mathtt{c}$, 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 $\{\mathtt{a}, \mathtt{b}, \mathtt{i}\}$ and $\{\mathtt{c}\}$ 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 [15, 26, 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. 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 $\{\mathtt{a}, \mathtt{b}, \mathtt{i}\}$ and $\{\mathtt{c}\}$, and make Octagon compute the following smaller matrix at the first query:

$$
\begin{array}{r|c|c|c|c|c|c}
 & \mathtt{a} & -\mathtt{a} & \mathtt{b} & -\mathtt{b} & \mathtt{i} & -\mathtt{i} \\
\hline
\mathtt{a} & 0 & \infty & 0 & \infty & -1 & \infty \\
\hline
-\mathtt{a} & \infty & 0 & \infty & 0 & \infty & \infty \\
\hline
\mathtt{b} & 0 & \infty & 0 & \infty & -1 & \infty \\
\hline
-\mathtt{b} & \infty & \infty & \infty & 0 & \infty & \infty \\
\hline
\mathtt{i} & \infty & \infty & \infty & \infty & 0 & \infty \\
\hline
-\mathtt{i} & \infty & -1 & \infty & -1 & \infty & \infty
\end{array}
\tag{2}
$$

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 [15]. 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 $\mathcal{F}$.

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 [15], 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:

$$\begin{array}{r|c|c|c|c|c|c|c|c}
 & \texttt{a} & -\texttt{a} & \texttt{b} & -\texttt{b} & \texttt{c} & -\texttt{c} & \texttt{i} & -\texttt{i} \\
\hline
\texttt{a} & \bigstar & \top & \bigstar & \top & \top & \top & \bigstar & \top \\
\hline
-\texttt{a} & \top & \bigstar & \top & \bigstar & \top & \top & \top & \top \\
\hline
\texttt{b} & \bigstar & \top & \bigstar & \top & \top & \top & \bigstar & \top \\
\hline
-\texttt{b} & \top & \bigstar & \top & \bigstar & \top & \top & \top & \top \\
\hline
\texttt{c} & \top & \top & \top & \top & \bigstar & \top & \top & \top \\
\hline
-\texttt{c} & \top & \top & \top & \top & \top & \bigstar & \top & \top \\
\hline
\texttt{i} & \top & \top & \top & \top & \top & \top & \bigstar & \top \\
\hline
-\texttt{i} & \top & \bigstar & \top & \bigstar & \top & \top & \top & \bigstar
\end{array} \tag{3}$$

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 $\mathtt{i}-\mathtt{b}$ and $\mathtt{b}-\mathtt{a}$, and this means that Octagon is likely to infer finite (thus informative) upper bounds of $\mathtt{i}-\mathtt{b}$ and $\mathtt{b}-\mathtt{a}$. 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 $Q = \{q_1, \ldots, q_k\}$ 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 $q'_1, \ldots, q'_l$, we collect the results $m^\sharp_1, \ldots, m^\sharp_l$ of the pre-analysis at these queries, and generate the following labeled data:

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

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.[4] This generation process is applied for all programs $P_1, \ldots, P_N$ 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 $T = \{(\mathtt{a},\mathtt{b}), (\mathtt{b},\mathtt{a}), (\mathtt{a},\mathtt{i}), (\mathtt{i},\mathtt{a}), (\mathtt{b},\mathtt{i}), (\mathtt{i},\mathtt{b}), (\mathtt{a},\mathtt{a}), (\mathtt{b},\mathtt{b}), (\mathtt{c},\mathtt{c}), (\mathtt{i},\mathtt{i})\}$.

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 [9].

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

[4] 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.

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:

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

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.

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 $(\mathbb{C}, \hookrightarrow)$, where $\mathbb{C}$ is the set of program points and $(\hookrightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ 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 [8]. 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

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

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

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

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

Fig. 2. 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 $\bot$. Also, in $x_i = x_j + k$, we consider only the case that $i \neq j$.

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 $\mathbb{Z} \cup \{\infty\}$, which means the following constraint:

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

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

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

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 [8].

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 2. 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 $c \in \mathbb{C}$):

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

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 [8, 15]. 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:

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

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:

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

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 express 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[5] 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 [15], 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 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 [15], 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

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

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

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

$$\llbracket x_i = ? \rrbracket^\sharp m^\sharp = m_1^\sharp \text{ when } (m_1^\sharp)_{pq} = \begin{cases} \top & (p \in \{2i-1, 2i\} \wedge q \notin \{2i-1, 2i\}) \\ \top & (p \notin \{2i-1, 2i\} \wedge q \in \{2i-1, 2i\}) \\ \bigstar & (p = q = 2i-1 \vee p = q = 2i) \\ ((m^\sharp)^\bullet)_{pq} & \textit{otherwise} \end{cases}$$

Fig. 3. Abstract semantics of some primitive commands in the impact pre-analysis. In $x_i = x_j + k$, we show only the case that $i \neq j$.

computed by Octagon are likely to contain nontrivial bounds. If the ij-th entry of $m^\sharp$ is $\bigstar$, the ij-th entry of m is likely to be 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 is a Galois connection between the powerset lattice $\wp(\mathbb{O})$ (with the usual subset order) and $\mathbb{O}^\sharp$:

$$\begin{aligned} &\gamma : \mathbb{O}^\sharp \to \wp(\mathbb{O}), && \gamma(m^\sharp) = \{\bot\} \cup \{m \in \mathbb{O} \mid \forall i, j.\ (m^\sharp_{ij} = \bigstar \implies m_{ij} \neq \infty)\}, \\ &\alpha : \wp(\mathbb{O}) \to \mathbb{O}^\sharp, && \alpha(M)_{ij} = \bigstar \iff (\bigsqcup M \neq \bot \implies (\bigsqcup M)_{ij} \neq \infty). \end{aligned}$$

The pre-analysis uses the abstract semantics $\llbracket \mathit{cmd} \rrbracket^\sharp : \mathbb{O}^\sharp \to \mathbb{O}^\sharp$ that is derived from this Galois connection and the abstract semantics of the same command in Octagon (Figure 2). Figure 3 shows the results of this derivation.

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:

$$\begin{aligned} Q_i' = \{c \mid \exists p.\,(c, p) \in Q_i \wedge{} & \text{the } jk \text{ entry is about the upper bound claimed in } p \\ & \wedge X_i(c) \neq \bot \wedge X_i(c)_{jk} = \bigstar\}, \\ \mathcal{D} = \bigcup_{1 \le i \le N} \{(P_i, (x_j, x_k), L) \mid{} & L = \oplus \iff \\ & \exists c \in Q_i'.\exists l, m \in \{0, 1\}.\, X_i(c)_{(2j-l)(2k-m)} = \bigstar\}. \end{aligned}$$

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 $\mathbf{f} = \{f_1, \ldots, f_m\}$, 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:

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

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 [9] 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 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,

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

Table 1. Features for relations of two variables.

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 $x = \texttt{malloc}(y)$ or $y = \texttt{malloc}(x)$.
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 $x[y]$ or $y[x]$.
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.

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

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

The partition $\mathit{\Pi}$ 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., $\mathcal{F}(P,(x,z)) = \ominus$), $\mathit{\Pi}$ may put x and z in the same $\pi \in \mathit{\Pi}$, if $\mathcal{F}(P,(x,y)) = \mathcal{F}(P,(y,z)) = \oplus$ for some y. Effectively, our construction of $\mathit{\Pi}$ 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 Experiments

We describe the experimental evaluation of our method for learning a variable-clustering strategy. The evaluation aimed to answer the following questions:

1. **Effectiveness**: How well does the partial Octagon 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 [25]. 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 [15]. In our experiments, we modified Sparrow to use the partial Octagon analysis as presented in Section 3, instead of Interval. The partial Octagon was implemented on top of the sparse analysis framework [14, 13], so it is significantly faster than the vanilla Octagon analysis [8]. For the implementation of data structures and abstract operations for Octagon, we tried the OptOctagons plugin [24] of the Apron framework [6]. For the decision tree learning, we used Scikit-learn [17]. We used 17 open-source benchmark programs (Table 2) and all the experiments were done on a Ubuntu machine with Intel Xeon clocked at 2.4GHz cpu and 192GB of main memory.

5.1 Effectiveness

We evaluated the effectiveness of a strategy learned by our method on the cost and precision of Octagon. We compared the partial Octagon analysis with a learned variable-clustering strategy with the Interval analysis and the approach for optimizing Octagon in [15]. The approach in [15] 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. Table 2 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

Table 2. 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. **LOC** reports lines of code before preprocessing. **Var** reports the number of program variables (more precisely, abstract locations). **#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 (**Ours**). **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.

			#Alarms			**Time(s)**		
Program	**LOC**	**Var**	**Itv**	**Impt**	**Ours**	**Itv**	**Impt**	**Ours**
brutefir-1.0f	103	54	4	0	0	0	0 (0)	0 (0)
consolcalculator-1.0	298	165	20	10	10	0	0 (0)	0 (0)
id3-0.15	512	527	15	6	6	0	0 (0)	1 (0)
spell-1.0	2,213	450	20	8	17	0	1 (1)	1 (0)
mp3rename-0.6	2,466	332	33	3	3	0	1 (0)	1 (0)
irmp3-0.5.3.1	3,797	523	2	0	0	1	2 (0)	3 (1)
barcode-0.96	4,460	1,738	235	215	215	2	9 (7)	6 (1)
httptunnel-3.3	6,174	1,622	52	29	27	3	35 (32)	5 (1)
e2ps-4.34	6,222	1,437	119	58	58	3	6 (3)	3 (0)
bc-1.06	13,093	1,891	371	364	364	14	252 (238)	16 (1)
less-382	23,822	3,682	625	620	625	83	2,354 (2,271)	87 (4)
bison-2.5	56,361	14,610	1,988	1,955	1,955	137	4,827 (4,685)	237 (79)
pies-1.2	66,196	9,472	795	785	785	49	14,942 (14,891)	95 (43)
icecast-server-1.3.12	68,564	6,183	239	232	232	51	109 (55)	107 (42)
raptor-1.4.21	76,378	8,889	2,156	2,148	2,148	242	17,844 (17,604)	345 (104)
dico-2.0	84,333	4,349	402	396	396	38	156 (117)	51 (24)
lsh-2.0.4	110,898	18,880	330	325	325	33	139 (106)	251 (218)
Total			7,406	7,154	7,166	656	40,677 (40,011)	1,207 (519)

7,406 alarms from the benchmark set.[6] The existing approach for partial Octagon [15] 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 [15]. For instance, for `spell-1.0`, our approach is less precise than that of [15] because some usage patterns of variables in `spell-1.0` do not appear in other programs. On the other hand, for `httptunnel-3.3`, our approach produces better results because the impact pre-analysis of [15] uses ★ conservatively and fails to identify some important relationships between variables.

[6] 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.

Table 3. Generalization performance.

			#Alarms			Time		
Program	**LOC**	**Var**	**Itv**	**All**	**Small**	**Itv**	**All**	**Small**
pies-1.2	66,196	9,472	795	785	785	49	95 (43)	98 (43)
icecast-server-1.3.12	68,564	6,183	239	232	232	51	113 (42)	99 (42)
raptor-1.4.21	76,378	8,889	2,156	2,148	2,148	242	345 (104)	388 (104)
dico-2.0	84,333	4,349	402	396	396	38	61 (24)	62 (24)
lsh-2.0.4	110,898	18,880	330	325	325	33	251 (218)	251 (218)
Total			3,922	3,886	3,886	413	864 (432)	899 (432)

5.2 Generalization

Although the impact pre-analysis scales far better than Octagon, it is still too expensive to be used routinely for large programs (> 100 KLOC). Therefore, in order for our approach to scale, the variable-clustering strategy learned from a codebase of small programs needs 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 a variable-clustering strategy from the group of small programs, and evaluated its performance on that of large programs.

Table 3 shows the results. Columns labeled **Small** report the performance of our approach learned from the small programs. **All** reports the performance of the strategy used in Section 5.1 (i.e., the strategy learned with all benchmark programs except for each target 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.

5.3 Feature Design

We identified top ten features that are most important to learn an effective variable-clustering strategy for Octagon. We applied our method to all the 17 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.

5.4 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 [10], 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 5.1.

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.

6 Related Work

The scalability issue of the Octagon analysis is well-known, and there have been various attempts to optimize the analysis [14, 24]. Oh et al. [14] exploited the data dependencies of a program and removed unnecessary propagation of information between program points during Octagon's fixpoint computation. Singh et al. [24] 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 [6]. 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 [14, 24] 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 [15]. 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 [15]. The approach based on the pre-analysis [15], 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.

Recently there have been a large amount of research activities for developing data-driven approaches to challenging program analysis problems, such as specification inference [20, 18], invariant generation [11, 19, 21–23, 4], acceleration of abstraction refinement [5], and smart report of analysis results [7, 12, 27]. In particular, Oh et al. [16] 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 [16], 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 5.2 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 [16], 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.

7 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 [15]. 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%.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. We also thank Kwangkeun Yi, Chung-Kil Hur, and all members of SoFA group members in Seoul National University for their helpful comments and suggestions. This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1502-07 and Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.R0190-16-2011, Development of Vulnerability Discovery Technologies for IoT Software Security). This research was also supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(NRF-2016R1C1B2014062).

References

1. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In *PLDI*, 2003.
2. Leo Breiman. Random Forests. *Machine Learning*, 2001.
3. Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL*, 1978.
4. Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *POPL*, pages 499–512, 2016.
5. Radu Grigore and Hongseok Yang. Abstraction refinement guided by a learnt probabilistic model. In *POPL*, 2016.
6. Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, 2009.
7. Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. In *ESEC/FSE*, pages 462–473, 2015.
8. Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 2006.
9. Tom M. Mitchell. *Machine learning*. McGraw-Hill, Inc., 1997.
10. K P Murphy. *Machine learning: a probabilistic perspective (adaptive computation and machine learning series)*. Mit Press ISBN, 2012.
11. Aditya V. Nori and Rahul Sharma. Termination proofs from tests. In *FSE*, pages 246–256, 2013.
12. Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *POPL*, pages 469–484, 2016.
13. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. Global sparse analysis framework. *ACM Trans. Program. Lang. Syst.*, 36(3):8:1–8:44, September 2014.
14. Hakjoo Oh, Kihong Heo, Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI*, 2012.

15. Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*, 2014.
16. Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *OOPSLA*, 2015.
17. Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research*, 2011.
18. Veselin Raychev, Pavol Bielik, Martin T. Vechev, and Andreas Krause. Learning programs from noisy data. In *POPL*, pages 761–774, 2016.
19. Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *ISSTA*, pages 295–306, 2008.
20. Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Mining library specifications using inductive logic programming. In *ICSE*, pages 131–140, 2008.
21. Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, pages 574–592, 2013.
22. Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. Verification as learning geometric concepts. In *SAS*, pages 388–411, 2013.
23. Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In *CAV*, pages 71–87, 2012.
24. Gagandeep Singh, Markus Püschel, and Martin Vechev. Making Numerical Program Analysis Fast. In *PLDI*, 2015.
25. Sparrow. `http://ropas.snu.ac.kr/sparrow`.
26. Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI*, 2004.
27. Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static C analyzer. *Information Processing Letters*, 102(2-3):118–123, 2007.