Learning a Strategy for Choosing Widening Thresholds from a Large Codebase

Sooyoung Cha, Sehun Jeong, and Hakjoo Oh

Korea University

Abstract. In numerical static analysis, the technique of widening thresholds is essential for improving the analysis precision, but blind uses of the technique often significantly slow down the analysis. Ideally, an analysis should apply the technique only when it benefits, by carefully choosing thresholds that contribute to the final precision. However, finding the proper widening thresholds is nontrivial and existing syntactic heuristics often produce suboptimal results. In this paper, we present a method that automatically learns a good strategy for choosing widening thresholds from a given codebase. A notable feature of our method is that a good strategy can be learned with analyzing each program in the codebase only once, which allows to use a large codebase as training data. We evaluated our technique with a static analyzer for full C and 100 open-source benchmarks. The experimental results show that the learned widening strategy is highly cost-effective; it achieves 84% of the full precision while increasing the baseline analysis cost only by 1.4x. Our learning algorithm is able to achieve this performance 26 times faster than the previous Bayesian optimization approach.

1 Introduction

In static analysis for discovering numerical program properties, the technique of widening with thresholds is essential for improving the analysis precision [1–4, 6–9]. Without the technique, the analysis often fails to establish even simple numerical invariants. For example, suppose we analyze the following code snippet with the interval domain:

```
1  i = 0;
2  while (i != 4) {
3    i = i + 1;
4    assert(i <= 4);
5  }
```

Note that the interval analysis with the standard widening operator cannot prove the safety of the assertion at line 4. The analysis concludes that the interval value of i right after line 2 is $[0, +\infty]$ (hence $[1, +\infty]$ at line 4) because of the widening operation applied at the entry of the loop. A simple way of improving the result is to employ widening thresholds. For example, when an integer 4 is used as a

threshold, the widening operation at the loop entry produces the interval $[0, 4]$, instead of $[0, +\infty]$, for the value of i. The loop condition $i \neq 4$ narrows down the value to $[0, 3]$ and therefore we can prove that the assertion holds at line 4.

However, it is a challenge to choose the right set of thresholds that improves the analysis precision with a small extra cost. Simple-minded methods can hardly be cost-effective. For example, simply choosing all integer constants in the program would not scale to large programs. Existing syntactic and semantics heuristics for choosing thresholds (e.g. [3, 9, 8, 6]) are also not satisfactory. For example, the syntactic heuristic used in [3], which is specially designed for the flight control software, is not precision-effective in general [12]. A more sophisticated, semantics-based heuristic sometimes incurs significant cost blow up [8]. No existing techniques are able to prescribe small yet effective set of thresholds for arbitrary programs.

In this paper, we present a technique that automatically learns a good strategy for choosing widening thresholds from a given codebase. The learned strategy is then used for analyzing new, unseen programs. Our technique includes a parameterized strategy for choosing widening thresholds, which decides whether to use each integer constant in the given program as a threshold or not. Following [13], the strategy is parameterized by a vector of real numbers and the effectiveness of the strategy is completely determined by the choice of the parameter. Therefore, in our approach, learning a good strategy corresponds to finding a good parameter from a given codebase.

A salient feature of our method is that a good strategy can be learned by analyzing the codebase only once, which enables us to use a large codebase as a training dataset. In [13], learning a strategy is formulated as a blackbox optimization problem and the Bayesian optimization approach was proposed to efficiently solve the optimization problem. However, we found that this approach is still too costly when the codebase is large, mainly because it requires multiple runs of the static analyzer over the entire codebase. Motivated by this limitation, we designed a new learning algorithm that does not require running the analyzer over the codebase multiple times. The key idea is to use an oracle that quantifies the relative importance of each integer constant in the program with respect to improving the analysis precision. With this oracle, we transform the blackbox optimization problem to a whitebox one that is much easier to solve than the original problem. We show that the oracle can be effectively obtained from a single run of the static analyzer over the codebase.

The experimental results show that our learning algorithm produces a highly cost-effective strategy and is fast enough to be used with a large codebase. We implemented our approach in a static analyzer for real-world C programs and used 100 open-source benchmarks for the evaluation. The learned widening strategy achieves 84% of the full precision (i.e., the precision of the analysis using all integer constants in the program as widening thresholds) while increasing the cost of the baseline analysis without widening thresholds only by 1.4x. Our learning algorithm is able to achieve this performance 26 times faster than the existing Bayesian optimization approach.

Contributions This paper makes the following contributions.

- We present a learning-based method for selectively applying the technique of widening thresholds. From a given codebase, our method automatically learns a strategy for choosing widening thresholds.
- We present a new, oracle-guided learning algorithm that is significantly faster than the existing Bayesian optimization approach. Although we use this algorithm for learning widening strategy, our learning algorithm is generally applicable to adaptive static analyses in general provided a suitable oracle is given for each analysis.
- We prove the effectiveness of our method in a realistic setting. Using a large codebase of 100 open-source programs, we experimentally show that our learning strategy is highly cost-effective, achieving the 84% of the full precision while increasing the cost by 1.4 times.

Outline We first present our learning algorithm in a general setting; Section 2 defines a class of adaptive static analyses and Section 3 explains our oracle-guided learning algorithm. Next, in Section 4, we describe how to apply the general approach to the problem of learning a widening strategy. Section 5 presents the experimental results, Section 6 discusses related work, and Section 7 concludes.

2 Adaptive Static Analysis

We use the setting of adaptive static analysis in [13]. Let $P \in \mathbb{P}$ be a program to analyze. Let $\mathbb{J}_P$ be a set of indices that represent parts of P. Indices in $\mathbb{J}_P$ are used as "switches" that determine whether to apply high precision or not. For example, in the partially flow-sensitive analysis in [13], $\mathbb{J}_P$ is the set of program variables and the analysis applies flow-sensitivity only to a selected subset of $\mathbb{J}_P$. In this paper, $\mathbb{J}_P$ denotes the set of constant integers in the program and our aim is to choose a subset of $\mathbb{J}_P$ that will be used as widening thresholds. Once $\mathbb{J}_P$ is chosen, the set $\mathcal{A}_P$ of program abstractions is defined as a set of indices as follows:

$$\mathbf{a} \in \mathcal{A}_P = \wp(\mathbb{J}_P).$$

In the rest of the paper, we omit the subscript P from $\mathbb{J}_P$ and $\mathcal{A}_P$ when there is no confusion.

The program is given together with a set of queries (i.e. assertions) and the goal of the static analysis is to prove as many queries as possible. We suppose that an adaptive static analysis is given with the following type:

$$F : \mathbb{P} \times \mathcal{A} \to \mathbb{N}.$$

Given a program P and its abstraction $\mathbf{a}$, the analysis $F(P, \mathbf{a})$ analyzes the program P by applying high precision (e.g. widening thresholds) only to the program parts in the abstraction $\mathbf{a}$. For example, $F(P, \emptyset)$ and $F(P, \mathbb{J}_P)$ represent the least and most precise analyses, respectively. The result from $F(P, \mathbf{a})$

indicates the number of queries in P proved by the analysis. We assume that the abstraction correlates the precision and cost of the analysis. That is, if $\mathbf{a}'$ is a more refined abstraction than $\mathbf{a}$ (i.e. $\mathbf{a} \subseteq \mathbf{a}'$), then $F(P, \mathbf{a}')$ proves more queries than $F(P, \mathbf{a})$ does but the former is more expensive to run than the letter. This assumption usually holds in program analyses for C.

In this paper, we are interested in automatically finding an adaptation strategy

$$\mathcal{S} : \mathbb{P} \to \mathcal{A}$$

from a given codebase $\mathbf{P} = \{P_1, \ldots, P_m\}$. Once the strategy is learned, it is used for analyzing unseen program P as follows:

$$F(P, \mathcal{S}(P)).$$

Our goal is to learn a cost-effective strategy $\mathcal{S}^*$ such that $F(P, \mathcal{S}^*(P))$ has precision comparable to that of the most precise analysis $F(P, \mathbb{J}_P)$ while its cost remains close to that of the least precise one $F(P, \emptyset)$.

3 Learning an Adaptation Strategy from a Codebase

In this section, we explain our method for learning a strategy $\mathcal{S} : \mathbb{P} \to \mathcal{A}$ from a codebase $\mathbf{P} = \{P_1, \ldots, P_m\}$. Our method follows the overall structure of the learning approach in [13] but uses a new learning algorithm that is much more efficient than the Bayesian optimization approach in [13].

In Section 3.1, we summarize the definition of the adaptation strategy in [13], which is parameterized by a vector $\mathbf{w}$ of real numbers. In Section 3.2, the optimization problem of learning is defined. Section 3.3 briefly presents the existing Bayesian optimization method for solving the optimization problem and discusses its limitation in performance. Finally, Section 3.4 presents our learning algorithm that avoids the problem of the existing approach.

3.1 Parameterized Adaptation Strategy

In [13], the adaptation strategy is parameterized and the result of the strategy is limited to a particular set of abstractions. That is, the parameterized strategy is defined with the following type:

$$\mathcal{S}_{\mathbf{w}} : \mathbb{P} \to \mathcal{A}^k$$

where $\mathcal{A}^k = \{\mathbf{a} \in \mathcal{A} \mid |\mathbf{a}| = k\}$ is the set of abstractions of size k. The strategy is parameterized by $\mathbf{w} \in \mathbb{R}^n$, a vector of real numbers. In this paper, we assume that k is fixed, which is set to 30 in our experiments, and $\mathbb{R}$ denotes real numbers between -1 and 1, i.e., $\mathbb{R} = [-1, 1]$. The effectiveness of the strategy is solely determined by the parameter $\mathbf{w}$. With a good parameter $\mathbf{w}$, the analysis $F(P, \mathcal{S}_{\mathbf{w}}(P))$ has precision comparable to the most precise analysis $F(P, \mathbb{J}_P)$ while its cost is not far different from the least precise one $F(P, \emptyset)$. Our goal is to learn a good parameter $\mathbf{w}$ from a codebase $\mathbf{P} = \{P_1, P_2, \ldots, P_m\}$.

The parameterized adaptation strategy $\mathcal{S}_{\mathbf{w}}$ is defined as follows. We assume that a set of program features is given:

$$\mathbf{f}_P = \{f_P^1, f_P^2, \ldots, f_P^n\}$$

where a feature f_P^k is a predicate over the switches $\mathbb{J}_P$:

$$f_P^k : \mathbb{J}_P \to \mathbb{B}.$$

In general, a feature is a function of type $\mathbb{J}_P \to \mathbb{R}$ but we assume that the result is binary for simplicity. Note that the number of features equals to the dimension of $\mathbf{w}$. With the features, a switch j is represented by a feature vector as follows:

$$\mathbf{f}_P(j) = \langle f_P^1(j), f_P^2(j), \ldots, f_P^n(j)\rangle.$$

The strategy $\mathcal{S}_{\mathbf{w}}$ works in two steps:

1. Compute the scores of switches. The score of switch j is computed by a linear combination of its feature vector and the parameter $\mathbf{w}$:

$$score_P^{\mathbf{w}}(j) = \mathbf{f}_P(j) \cdot \mathbf{w}. \tag{1}$$

   The score of an abstraction $\mathbf{a}$ is defined by the sum of the scores of elements in $\mathbf{a}$:

$$score_P^{\mathbf{w}}(\mathbf{a}) = \sum_{j \in \mathbf{a}} score_P^{\mathbf{w}}(j).$$

2. Select the top-k switches. Our strategy selects top-k switches with highest scores:

$$\mathcal{S}_{\mathbf{w}}(P) = \operatorname*{argmax}_{\mathbf{a} \in \mathcal{A}_P^k} score_P^{\mathbf{w}}(\mathbf{a}).$$

3.2 The Optimization Problem

Learning a good parameter $\mathbf{w}$ from a codebase $\mathbf{P} = \{P_1, \ldots, P_m\}$ corresponds to solving the following optimization problem:

$$\text{Find } \mathbf{w}^* \in \mathbb{R}^n \text{ that maximizes } obj(\mathbf{w}^*) \tag{2}$$

where the objective function is

$$obj(\mathbf{w}) = \sum_{P_i \in \mathbf{P}} F(P_i, \mathcal{S}_{\mathbf{w}}(P_i)).$$

That is, we aim to find a parameter $\mathbf{w}^*$ that maximizes the number of queries in the codebase that are proved by the static analysis with $\mathcal{S}_{\mathbf{w}^*}$. Note that it is only possible to solve the optimization problem approximately because the search space is very large. Furthermore, evaluating the objective function is typically very expensive since it involves running the static analysis over the entire codebase.

3.3 Existing Approach

In [13], a learning algorithm based on Bayesian optimization has been proposed. To simply put, this algorithm performs a random sampling guided by a probabilistic model:

```
1: repeat
2:     sample w from R^n using probabilistic model M
3:     s ← obj(w)
4:     update the model M with (w, s)
5: until timeout
6: return best w found so far
```

The algorithm uses a probabilistic model $\mathcal{M}$ that approximates the objective function by a probabilistic distribution on function spaces (using the Gaussian Process [14]). The purpose of the probabilistic model is to pick a next parameter to evaluate that is predicted to work best according the approximation of the objective function (line 2). Next, the algorithm evaluates the objective function with the chosen parameter $\mathbf{w}$ (line 3). The model $\mathcal{M}$ gets updated with the current parameter and its evaluation result (line 4). The algorithm repeats this process until the cost budget is exhausted and returns the best parameter found so far.

Although this algorithm is significantly more efficient than the random sampling [13], it still requires a number of iterations of the loop to learn a good parameter. According to our experience, the algorithm with Bayesian optimization typically requires more than 100 iterations to find good parameters (Section 5). Note that even a single iteration of the loop can be very expensive in practice because it involves running the static analyzer over the entire codebase. When the codebase is massive and the static analyzer is costly, evaluating the objective function multiple times is prohibitively expensive.

3.4 Our Oracle-Guided Approach

In this paper, we present a method for learning a good parameter without analyzing the codebase multiple times. By analyzing each program in the codebase only once, our method is able to find a parameter that is as good as the parameter found by the Bayesian optimization method.

We achieve this by applying an *oracle-guided* approach to learning. Our method assumes the presence of an oracle $\mathcal{O}_P$ for each program P, which maps program parts in $\mathbb{J}_P$ to real numbers in $\mathbb{R} = [-1, 1]$:

$$\mathcal{O}_P : \mathbb{J}_P \to \mathbb{R}.$$

For each $j \in \mathbb{J}_P$, the oracle returns a real number that quantifies the relative contribution of j in achieving the precision of $F(P, \mathbb{J}_P)$. That is, $\mathcal{O}(j_1) < \mathcal{O}(j_2)$ means that j_2 contributes more than j_1 to improving the precision during the analysis of $F(P, \mathbb{J}_P)$. We assume that the oracle is given together with the adaptive static analysis. In Section 4.3, we show that such an oracle easily results from analyzing the program for interval analysis with widening thresholds.

In the presence of the oracle, we can establish an easy-to-solve optimization problem which serves as a proxy of the original optimization problem in (2). For simplicity, assume that the codebase consists of a single program: $\mathbf{P} = \{P\}$. Shortly, we extend the method to multiple training programs. Let $\mathcal{O}$ be the oracle for program P. Then, the goal of our method is to learn $\mathbf{w}$ such that, for every $j \in \mathbb{J}_P$, the scoring function in (1) instantiated with $\mathbf{w}$ produces a value that is as close to $\mathcal{O}(j)$ as possible. We formalize this optimization problem as follows:

$$\text{Find } \mathbf{w}^* \text{ that minimizes } E(\mathbf{w}^*)$$

where $E(\mathbf{w})$ is defined to be the *mean square error* of $\mathbf{w}$:

$$\begin{aligned} E(\mathbf{w}) &= \sum_{j \in \mathbb{J}_P} (score_P^{\mathbf{w}}(j) - \mathcal{O}(j))^2 \\ &= \sum_{j \in \mathbb{J}_P} (\mathbf{f}_P(j) \cdot \mathbf{w} - \mathcal{O}(j))^2 \\ &= \sum_{j \in \mathbb{J}_P} (\sum_{i=1}^{n} f_P^i(j)\mathbf{w}_i - \mathcal{O}(j))^2. \end{aligned}$$

Note that the body of the objective function $E(\mathbf{w})$ is a differentiable, closed-form expression, so we can use the standard gradient decent algorithm to find a minimum of E. The algorithm is simply stated as follows:

1: sample $\mathbf{w}$ from $\mathbb{R}^n$
2: **repeat**
3: $\quad \mathbf{w} = \mathbf{w} - \alpha \cdot \nabla E(\mathbf{w})$
4: **until** convergence
5: **return w**

Starting from a random parameter $\mathbf{w}$ (line 1), the algorithm keeps going down toward the minimum in the direction against the gradient $\nabla E(\mathbf{w})$. The single step size is determined by the learning rate α. The gradient of E is defined as follows:

$$\nabla E(\mathbf{w}) = \big(\frac{\partial}{\partial \mathbf{w}_1} E(\mathbf{w}), \frac{\partial}{\partial \mathbf{w}_2} E(\mathbf{w}), \cdots, \frac{\partial}{\partial \mathbf{w}_n} E(\mathbf{w})\big)$$

where the partial derivatives are

$$\frac{\partial}{\partial \mathbf{w}_k} E(\mathbf{w}) = 2 \sum_{j \in \mathbb{J}_P} (\sum_{i=1}^{n} f_P^i(j)\mathbf{w}_i - \mathcal{O}(j)) f_P^k(j)$$

Because the optimization problem does not involve the static analyzer and codebase, learning a parameter $\mathbf{w}$ is done quickly regardless of the cost of the analysis and the size of the codebase, and in the next section, we show that a good-enough oracle can be obtained by analyzing the codebase only once.

It is easy to extend the method to multiple programs. Let $\mathbf{P} = \{P_1, \ldots, P_m\}$ be the codebase. We assume the presence of oracles $\mathcal{O}_{P_1}, \ldots, \mathcal{O}_{P_m}$ for each program $P_i \in \mathbf{P}$. We establish the error function $E_{\mathbf{P}}$ over the entire codebase as

follows:

$$E_{\mathbf{P}}(\mathbf{w}) = \sum_{P \in \mathbf{P}} \sum_{j \in \mathbb{J}_P} (\sum_{i=1}^{n} f_P^i(j)\mathbf{w}_i - \mathcal{O}_P(j))^2$$

and now the gradient $\nabla E_{\mathbf{P}}(\mathbf{w})$ is defined with the partial derivatives:

$$\frac{\partial}{\partial \mathbf{w}_k} E_{\mathbf{P}}(\mathbf{w}) = 2 \sum_{P \in \mathbf{P}} \sum_{j \in \mathbb{J}_P} (\sum_{i=1}^{n} f_P^i(j)\mathbf{w}_i - \mathcal{O}(j)) f_P^k(j).$$

Again, we use the gradient decent algorithm to find $\mathbf{w}$ that minimizes $E_{\mathbf{P}}(\mathbf{w})$.

4 Learning a Strategy for Widening Thresholds

In this section, we explain how to employ the oracle-guided method to learn a widening threshold strategy from a codebase. In Section 4.1, we define an interval analysis that uses widening with thresholds. Section 4.2 and 4.3 present the features and oracle that we used for the interval analysis, respectively.

4.1 Interval Analysis with Widening Thresholds

We assume that a program $P \in \mathbb{P}$ is represented by a control flow graph $P = (\mathbb{C}, \hookrightarrow)$, where $\mathbb{C}$ is the set of nodes (i.e. program points) and $(\hookrightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ is a binary relation denoting control-flows of the program; $c' \to c$ means that c is the program point next to c'.

The abstract domain of the analysis maps programs points to abstract states:

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

where $\mathbb{S}$ is a map from program variables to the interval domain:

$$\mathbb{S} = \mathit{Var} \to \mathbb{I}.$$

The abstract semantic function of the analysis is defined as follows:

$$F(X) = \lambda c.\ f_c(\bigsqcup_{c' \to c} X(c'))$$

where we assume that transfer function $f_c : \mathbb{S} \to \mathbb{S}$ is defined for each command c. The goal of the analysis is to compute an upper bound of the least fixed point of F:

$$\mathsf{lfp} F = \bigsqcup_{i \geq 0} F^i(\bot) = F^0(\bot) \sqcup F^1(\bot) \sqcup F^2(\bot) \sqcup \cdots$$

This fixed point iteration may not terminate because the interval domain $\mathbb{I}$ is of infinite height. Therefore, the analysis should use a widening operator for $\mathbb{I}$. A simple widening operator for the interval domain can be defined as follows: (For simplicity, we omit the cases when intervals are bottom).

$$[l_1, u_1] \triangledown [l_2, u_2] = [(l_2 < l_1 ? -\infty : l_1), (u_1 < u_2 ? +\infty : u_1)] \tag{3}$$

Note that this widening operator is very hasty and immediately replaces unstable bounds by ∞.

The technique of widening with thresholds aims to improve the precision by bounding the extrapolation by widening. Suppose we have a set $T \subseteq \mathbb{Z}$ of thresholds. These thresholds are successively used as a candidate of a fixed point. Formally, the widening operator $\triangledown_T$ with thresholds is defined as follows:

$$[l_1, u_1] \triangledown_T [l_2, u_2] = [(l_2 < l_1 ? glb(T, l_2) : l_1), (u_1 < u_2 ? lub(T, u_2) : u_1)] \quad (4)$$

where $glb(T, i)$ and $lub(T, i)$ are respectively the greatest lower bound and least upper bound of i in thresholds T:

$$glb(T, i) = \max\{n \in T \mid n \leq i\}$$
$$lub(T, i) = \min\{n \in T \mid n \geq i\}$$

The widening operators for $\mathbb{S}$ and $\mathbb{D}$ are defined pointwise.

The precision improvement by widening with thresholds crucially depends on the choice of the set T of thresholds, and our goal is to automatically learn a good strategy for choosing T from a given codebase. In our implementation, the set $\mathbb{J}_P$ in Section 5.1 corresponds to the set of all integer constants in program P, and the strategy $\mathcal{S}_{\mathbf{w}}$ chooses top-k integers from P based on the parameter $\mathbf{w}$.

4.2 Features

To use the learning algorithm, we need to design a set of features for integer constants in the program. We have designed 17 syntactic, semantic, and numerical features (Table 1). A feature is a predicate over integers. For example, the first feature in Table 1 indicates whether the number is used as the size of a statically allocated array in the program.

The features have been designed with simplicity and generality in mind. They do not depend on the interval analysis and therefore can be easily reused for other types of numerical analyses. Features 1–12 describe simple syntactic and semantic features for usages of integers in typical C programs. We used a flow-insensitive pre-analysis to extract the semantic features (e.g. feature 7). Features 13–17 describe numerical properties that are commonly found in C programs. We were curious whether these common numerical properties have impacts on the analysis precision when they are used for widening thresholds. Once these features are manually designed, it is the learning algorithm's job to decide how much they are relevant in the given analysis task.

4.3 Oracle

To use our new learning algorithm, we need the oracle:

$$\mathcal{O}_P : \mathbb{Z}_P \to \mathbb{R}$$

Table 1. Features for integer constants in C programs. Each feature represents a predicate over integers.

#	Description
1	used as the size of a static array
2	the size of a static array $-$ 1
3	returned by a function (e.g. return 1)
4	three successive numbers appear in the program (e.g. $n, n+1, n+2$)
5	most frequently appeared numbers in the program (i.e. top 10%)
6	least frequently appeared numbers in the program (i.e. bottom 10%)
7	passed as the size arguments of memory copy functions (e.g. memcpy)
8	used as the size of the destination arrays in memory copy functions (e.g. memcpy)
9	the null position of a string buffer involved in some loop condition
10	the null position of a static array of primitive types (e.g., arrays of int and char)
11	the null position of a static array of structure fields
12	constants involved in conditional expressions (e.g. if (x == 1))
13	integers of the form 2^n (e.g. 2, 4, 8, 16)
14	integers of the form $2^n - 1$ (e.g., 1, 3, 7, 15)
15	integers in the range $0 < n \leq 50$
16	integers in the range $50 < n \leq 100$
17	integers in the range $n > 1000$

where $\mathbb{Z}_P$ is the set of integer constants that appear in the program P. That is, $\mathcal{O}_P$ maps integer constants in the program into their relative importance when they are used for widening thresholds.

We use a simple heuristic to build the oracle. The idea is to analyze the codebase with full precision and estimate the importance by measuring how many times each integer constant contributes to stabilizing the fixed point computation. The term *full precision* means that the heuristic uses a thresholds set, which includes constant integers of the program's variables, the sizes of static arrays, and the lengths of constant strings. Through relatively cheap analysis (e.g., flow insensitive), we get an abstract memory state which holds the candidate thresholds information we mentioned above.

Let P be a program in the codebase. We analyze the program by using all its integer constants as thresholds. During the fixed point computation of the analysis, we observe each widening operation and maintain a map $\mathcal{C} : \mathbb{Z}_P \to \mathbb{N}$ that counts the integer constants involved in a local fixed point. That is, $\mathcal{C}(n)$ is initially 0 for all n, and whenever we perform the widening operation on intervals:

$$[l_1, u_1] \triangledown [l_2, u_2] = [l_3, u_3]$$

we check if the result reaches a local fixed point (i.e. $[l_3, u_3] \sqsubseteq [l_1, u_1]$). If so, we increase the counter values for l_3 and u_3: $\mathcal{C}(l_3) := \mathcal{C}(l_3)+1$ and $\mathcal{C}(u_3) := \mathcal{C}(u_3)+1$. We keep updating the counter $\mathcal{C}$ until a global fixd point is reached. Finally, we normalize the values in $\mathcal{C}$ to obtain the oracle $\mathcal{O}_P$. We repeat this process over the entire codebase and generate a set of oracles.

5 Experiments

In this section, we evaluate our approach with an interval analyzer for C and open-source benchmarks. We organized the experiments to answer the following research questions:

1. **Effectiveness**: How much is the analyzer with the learned strategy better than the baseline analyzers? (Section 5.2)
2. **Comparison**: How much is our learning algorithm better than the existing Bayesian optimization approach? (Section 5.3)
3. **Important Features**: What are the most important features identified by the learning algorithm? (Section 5.4)

5.1 Setting

We implemented our approach in Sparrow, a static buffer-overflow analyzer for real-world C programs [18]. The analysis is based on the interval abstract domain and performs a flow-sensitive and selectively context-sensitive analysis [11]. Along the interval analysis, it also simultaneously performs a flow-sensitive pointer analysis to handle indirect assignments and function pointers in C. The analyzer takes as arguments a set of integers to use for widening thresholds. Our technique automatically generates this input to the analyzer, by choosing a subset of integer constants that appear in the program.

To evaluate our approach, we collected 100 open-source C programs from GNU and Linux packages. The list of programs we used is available in Table 5. We randomly divided the 100 benchmark programs into 70 training programs and 30 testing programs. A strategy for choosing widening threshold is learned from the 70 training programs, and tested on the remaining 30 programs. We iterated this process for five times. Table 2 and 3 show the result of each trial. In our approach, based on our observation that the number of effective widening thresholds in each program is very small, we set k to 30, which means that the strategy chooses the top 30 integer constants from the program to use for widening thresholds.

In the experiments, we compared the performance of three analyzers.

- NoThld is the baseline Sparrow without widening thresholds. That is, it performs the interval analysis with the basic widening operator in (3).
- FullThld is a variant of Sparrow that uses all the integer constants in the program as widening thresholds. The thresholds set includes constant integers in the program, the sizes of static arrays, and the lengths of constant strings.
- Ours is our analyzer whose threshold strategy is learned from the codebase. That is, the threshold argument of the analyzer is given by the strategy learned from the 70 programs via our oracle-guided learning algorithm.

Table 2. Performance on the training programs.

Trial	Training			
	NoThld	FullThld	Ours	
	prove	prove	prove	quality
1	13,297	14,806	14,518	80.9%
2	14,251	15,912	15,602	81.3%
3	14,509	16,285	15,988	83.2%
4	11,931	13,313	13,020	78.8%
5	14,568	16,292	15,948	80.0%
TOTAL	68,556	76,608	75,076	**81.0%**

Table 3. Performance on the testing programs.

Trial	Testing								
	NoThld		FullThld			Ours			
	prove	sec	prove	sec	cost	prove	sec	quality	cost
1	5,083	222	5,785	1,789	8.0 x	5,637	361	78.9%	1.6 x
2	4,129	605	4,679	2,645	4.4 x	4,623	748	89.8%	1.2 x
3	3,871	397	4,306	1,068	2.7 x	4,237	543	84.1%	1.4 x
4	6,449	792	7,278	4,606	5.8 x	7,133	1228	82.5%	1.6 x
5	3,812	281	4,299	1,014	3.6 x	4,247	389	89.3%	1.4 x
TOTAL	23,344	2,297	26,347	11,122	**4.8 x**	25,877	3,269	**84.3%**	**1.4 x**

5.2 Effectiveness

Table 2 and 3 show the effectiveness of the learned strategy in the training and testing phases, respectively. Table 2 shows the training performance with 70 programs. For the five trials, NoThld proved 68,556 buffer-overrun queries. On the other hand, FullThld proved 76,608 queries. For the training programs, our learning algorithm was able to find a strategy that can prove 81.0% of the FullThld-only provable queries.

Table 3 shows the results on the 30 testing programs. In total, NoThld proved the 23,344 queries, while FullThld proved 26,347 queries. Our analysis with the learned strategy (Ours) proved 25,877 queries, achieving 84.3% of the precision of FullThld. In doing so, Ours increases the analysis time of NoThld only 1.4x, while FullThld increases the cost by 4.8x.

5.3 Comparison

We have implemented the previous learning algorithm based on Bayesian optimization [13] and compared its performance with that of our learning algorithm. Table 4 shows the results. For the five trials, our approach took on average 6,154 seconds to find a strategy of the average quality 81.0%. On the other hand, the

Table 4. Performance comparison with the Bayesian optimization approach. For Bayesian optimization, we set the maximum number of iterations to 100.

Trial	Learning Cost				
	OURS		Bayesian optimization		speedup
	quality	sec	quality	sec	
1	80.9%	6,682	74.3%	185,825	27.8 x
2	81.3%	5,971	80.1%	155,438	26.0 x
3	83.2%	7,192	77.1%	170,311	23.7 x
4	78.8%	3,976	73.7%	113,738	28.6 x
5	80.0%	6,947	74.7%	185,375	26.7 x
TOTAL	81.0%	30,768	76.0%	810,687	**26.3 x**

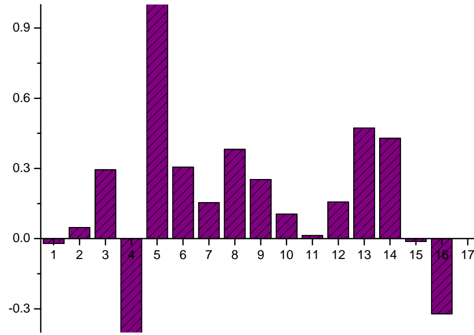

Fig. 1. Relative importance among features

Bayesian optimization approach was able to find a strategy that resulted 76.0% quality on training sets after it exhausted its iteration budget, which took on average 162,137 seconds. The results show that our learning algorithm is able to find a better strategy 26 times faster than the existing algorithm.

The Bayesian optimization approach did not work well with a limited time budget. When we allowed the Bayesian optimization approach to use the same time budget as ours, the existing approach ended up with a strategy of the average quality 57%. Note that our algorithm achieves the quality 81% in the same amount of time.

5.4 Important Features

In our approach, the learned parameter **w** indicates the relative importance of the features in Table 1. To identify the important features for widening thresholds,

we performed the training phase ten times and averaged the parameters obtained from each run.

Figure 1 shows the relative feature importance identified by the learning algorithm. During the ten trials, the feature 5 (most frequently appeared numbers in the program) was always the highest ranked feature. Features 13 (numbers of the form 2^n) and 14 (numbers of the form $2^n - 1$) were also consistently listed in the top 5.

These results were not expected from the beginning. At the initial stage of this work, we manually identified important features for widening thresholds and conjectured that the features 9, 10, and 11, which are related to null positions, are the most important ones. Consider the following code:

```
char *text="abcd";
i=0;
while (text[i] != NULL) {
   i++;
   assert(i <= 4);
}
```

When we convert the loop condition into an equivalent one $i \neq 4$ and use the null position 4 as a widening threshold, we can prove the safety of the assertion with the interval domain. We observed the above code pattern multiple times in the target programs being investigated and thought that using null position as thresholds would be one of the most important. However, the learning algorithm let us realize that unexpected features such as 5, 13, and 14 are the most important over the entire codebase, which is an insight hardly obtained manually because it is infeasible for humans to investigate the large codebase.

6 Related Work

Widening with Thresholds The technique of widening with thresholds has been widely used in numerical program analyses [7, 1–4, 9, 8, 6]. For example, its effectiveness has been shown with polyhedra [6], octagons [1, 3, 4], and intervals [7]. However, existing techniques use a fixed strategy for choosing the threshold set. For example, in [1, 3, 4, 7], all the integer constants that appear in conditional statements are used for the candidate of thresholds. In [6], a simple pre-analysis is used to infer a set of thresholds. The main limitation of these approaches is that the strategies are fixed and overfitted to some particular class of programs. For example, the syntactic and semantic heuristics were shown to be not always cost-effective [6, 7]. On the other hand, the goal of this paper is not to fix a particular strategy beforehand but to automatically learn a strategy from a given codebase, so that it can be adaptively used in practice.

Learning-based Program Analysis Recently, machine learning techniques are increasingly used in the field of program analysis [15–17, 13, 5, 10]. Among them,

our work lies in the direction of designing an adaptive static analysis via learning [13, 5]. In particular, our work is motivated by [13]'s result, which used Bayesian optimization to guide the learning process to more promising directions. We followed the general idea of the previous work, but we proposed a more efficient learning algorithm than the Bayesian optimization method. Because Oh et al.'s work uses the number of proven queries to measure quality of the learned strategy, the learning algorithm has to perform full-scale analysis on all training programs repeatedly until the learnt strategy meets a target quality. As we mentioned in Sec. 5.3, its takes too much time to get an acceptably good strategy over the large codebase. By contrast, our method reduces the learning cost by exploiting of the existence of the oracle for a given training program. Since the process of obtaining the oracle requires performing single full-scale analysis per training program, our learning algorithm radically reduced time cost than the existing method.

7 Conclusion

In this paper, we proposed a method that automatically learns a good strategy for choosing widening thresholds from a large codebase. We showed that the learned strategy is highly cost-effective; we can achieve 84% of the full precision with the 1.4x increase in analysis time.

The success of the method is largely attributed to our new learning algorithm that is significantly faster than the previous Bayesian optimization algorithm. In the presence of a large codebase, the Bayesian optimization approach failed to learn a good strategy in a reasonable amount of time. By contrast, our new learning algorithm is at least 26 times faster and is able to find a better parameter than the previous method.

Our approach is general enough to be used for other types of adaptive static analyses. As future work, we plan to apply our technique to other instances such as selective flow-sensitivity and context-sensitivity.

Acknowledgement. This work was supported by the Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.R0190-15-2011, Development of Vulnerability Discovery Technologies for IoT Software Security); the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B2014062); and the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2016-H85011610120001002) supervised by the IITP(Institute for Information & communications Technology Promotion).

References

1. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and imple-

mentation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, pages 85–108. Springer, 2002.
2. Olivier Bouissou, Yassamine Seladji, and Alexandre Chapoutot. Acceleration of the abstract fixpoint computation in numerical program analysis. *Journal of Symbolic Computation*, 47(12):1479 – 1511, 2012. International Workshop on Invariant Generation.
3. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Miné Antoine, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.
4. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In *Advances in Computer Science-ASIAN 2006. Secure Software and Related Issues*, pages 272–300. Springer, 2006.
5. Radu Grigore and Hongseok Yang. Abstraction refinement guided by a learnt probabilistic model. In *POPL*, 2016.
6. Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. In *FORMAL METHODS IN SYSTEM DESIGN*, pages 157–185, 1997.
7. Sol Kim, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Widening with thresholds via binary search. *Software: Practice and Experience*, 2015.
8. Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. Widening with thresholds for programs with complex control graphs. In *Proceedings of the 9th International Conference on Automated Technology for Verification and Analysis*, ATVA'11, pages 492–502, Berlin, Heidelberg, 2011. Springer-Verlag.
9. Bogdan Mihaila, Alexander Sepp, and Axel Simon. Widening as abstract domain. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, pages 170–184, 2013.
10. Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. Abstractions from tests. In *POPL*, 2012.
11. Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*, 2014.
12. Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective X-sensitive analysis guided by impact pre-analysis. *ACM Trans. Program. Lang. Syst.*, 38(2):6:1–6:45, December 2015.
13. Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via Bayesian optimisation. In *OOPSLA*, 2015.
14. Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.
15. Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, 2013.
16. Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. Verification as learning geometric concepts. In *SAS*, 2013.
17. Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In *CAV*, 2012.
18. Sparrow. `http://ropas.snu.ac.kr/sparrow`.

Table 5. Benchmark programs

Programs	LOC	Programs	LOC
wwl-1.3+db.c	474	e2ps-4.34.c	6,222
gosmore-0.0.0.20100711.c	497	apng2gif-1.5.c	6,522
ircmarkers-0.14.c	619	isdnutils-3.25+dfsg1.c	6,609
rovclock-0.6e.c	1,177	bwm-ng-0.6.c	6,833
xcircuit-3.7.55.dfsg.c	1,222	diffstat-1.58.c	7,077
iputils-20121221.c	1,311	lgrind-3.67.c	7,363
confget-1.02.c	1,393	lacheck-1.26.c	7,385
codegroup-19981025.c	1,518	lakai-0.1.c	7,487
time-1.7.c	1,759	libdebug-0.4.4.c	7,645
rexima-1.4.c	1,843	cmigemo-1.2+gh0.20140306.c	7,729
xinit-1.3.2.c	1,893	barcode-0.96.c	7,901
nlkain-1.3.c	1,927	apngopt-1.2.c	8,315
xchain-1.0.1.c	1,955	makedepf90-2.8.8.c	8,415
display-dhammapada-1.0.c	2,007	mpage-2.5.6.c	8,538
authbind-2.1.1.c	2,041	stripcc-0.2.0.c	8,914
unhtml-2.3.9.c	2,057	photopc-3.05.c	9,266
elfrc-0.7.c	2,142	psmisc-22.20.c	9,624
jbofihe-0.38.c	2,182	ircd-ircu-2.10.12.10.dfsg1.c	10,206
delta-2006.08.03.c	2,273	auto-apt-0.3.23ubuntu0.14.04.1.c	11,110
petris-1.0.1.c	2,411	glhack-1.2.c	11,237
libixp-0.6 20121202+hg148.c	2,428	sac-1.9b5.c	11,999
whichman-2.4.c	2,493	dict-gcide-0.48.1.c	12,318
acpi-1.7.c	2,597	gzip-spec2000.c	12,980
zmakebas-1.2.c	2,606	cutils-1.6.c	14,122
forkstat-0.01.04.c	2,710	mtr-0.85.c	14,127
setbfree-0.7.5.c	2,929	rhash-1.3.1.c	14,352
haskell98-tutorial-200006-2.c	3,161	gnuspool-1.7ubuntu1.c	16,665
kcc-2.3.c	3,429	smp-utils-0.97.c	17,520
ipip-1.1.9.c	3,605	ccache-3.1.9.c	17,536
gif2apng-1.7.c	3,816	gzip-1.2.4a.c	18,364
desproxy-0.1.0 pre3.c	3,841	netkit-ftp-0.17.c	19,254
magicfilter-1.2.c	3,856	libchewing-0.3.5.c	19,262
pgpgpg-0.13.c	3,908	archimedes.c	19,559
rsrce-0.2.2.c	3,956	tcs-1.c	19,967
rinetd-0.62.c	4,123	gnuplot-4.6.4.c	20,306
unsort-1.1.2.c	4,290	phalanx-22+d051004.c	24,099
hexdiff-0.0.53.c	4,334	gnuchess-5.05.c	28,853
acorn-fdisk-3.0.6.c	4,450	combine-0.3.3.c	29,508
pmccabe-2.6.c	4,920	rtai-3.9.1.c	30,739
dvbtune-0.5.ds.c	5,068	gnushogi-1.4.1.c	31,796
bmf-0.9.4.c	5,451	tmndec-3.2.0.c	31,890
libbind-6.0.c	5,497	fondu-0.0.20060102.c	32,298
mixal-1.08.c	5,570	libart-lgpl-2.3.21.c	38,815
cmdpack-1.03.c	5,575	flex-2.5.39.c	39,977
picocom-1.7.c	5,613	fwlogwatch-1.2.c	46,601
xdms-1.3.2.c	5,614	chrony-1.29.c	49,119
cifs-utils-6.0.c	5,815	uudeview-0.5.20.c	54,853
dtaus-0.9.c	6,018	sn-0.3.8.c	56,227
device-tree-compiler-1.4.0+dfsg.c	6,033	shadow-4.1.5.1.c	85,201
buildtorrent-0.8.c	6,170	skyeye-1.2.5.c	85,905