A Scalable Learning Algorithm for Data-Driven Program Analysis

Sooyoung Cha, Sehun Jeong, Hakjoo Oh*

Department of Computer Science and Engineering, Korea University, Seoul, Republic of Korea

Abstract

Context. : Recently data-driven program analysis has emerged as a promising approach for building cost-effective static analyzers. The ideal static analyzer should apply accurate but costly techniques only when they benefit. However, designing such a strategy for real-world programs is highly nontrivial and requires labor-intensive work. The goal of data-driven program analysis is to automate this process by learning the strategy from data through a learning algorithm.

Objective. : Current learning algorithms for data-driven program analysis are not scalable enough to be used with large codebases. The objective of this paper is to overcome this shortcoming and present a new algorithm that is able to efficiently learn a strategy from large codebases.

Method. : The key idea is to use an oracle and transform the existing blackbox learning problem into a whitebox one that is much easier to solve. The oracle quantifies the relative importance of each part of the program with respect to the analysis precision. The oracle can be obtained by running the most and least precise analyses only once over the codebase.

Results. : Our learning algorithm is much faster than the existing algorithms while producing high quality strategies. The evaluation is done with 140 open-source C programs, comprising of 2.1 MLoC in total. Learning at this large scale was previously impractical.

Conclusion. : Our work advances the state-of-the-art of data-driven program analysis by addressing the scalability issue of the existing learning algorithm. Our technique will make the data-driven approach more practical in the real-world.

Keywords: Data-driven program analysis, Learning algorithm

1. Introduction

The difficulty of building a cost-effective static analyzer has been a major challenge in program analysis [26, 25, 24, 8, 10, 16, 20, 27, 14, 23, 7]. Ideally, a static analyzer should be able to apply precision-improving but costly techniques only when it benefits the final analysis results. For example, techniques such as context-sensitivity [20, 23, 14] and relational analysis [5, 11] are expensive and therefore they must be used only for carefully selected procedures and variables. Traditionally, these decisions (e.g., which procedures to analyze context-sensitively) have been made with hand-crafted strategies [20, 23, 5, 17, 2]. However, manually designing such a strategy is labor-intensive and at risk of being suboptimal in practice.

Recently, data-driven program analysis has emerged as a promising approach to address this challenge [21, 3, 11, 12, 4, 13]. Instead of manually designing an analysis strategy, the approach aims to automatically generate the strategy from data. The learned strategy is then used for analyzing new, unseen programs. To this end, the approach parameterizes the strategy, which decides when and where to apply expensive techniques, and finds good parameter values from codebases through a learning algorithm. This approach has proven effective at automatically generating a variety of analysis strategies [21, 3, 11, 12] and the outcomes are likely to outperform manually-tuned strategies [13].

However, the current techniques for data-driven program analysis have a limitation that their learning algorithms do not scale to large codebases. In data-driven program analysis, the problem of learning parameters is formulated as a blackbox optimization problem whose objective function involves running a static analyzer over codebases [21, 13]. Solving this optimization problem is typically too expensive to be used with large codebases because it requires multiple runs of the static analyzer over the codebases. For instance, the learning algorithm with

*Corresponding author. Tel.: (+82 2 3290 4601)

Email addresses: sooyoungcha@korea.ac.kr (Sooyoung Cha), gifaranga@korea.ac.kr (Sehun Jeong), hakjoo_oh@korea.ac.kr (Hakjoo Oh)

Bayesian optimization [21] required more than 24 hours to learn good parameter values over 20 medium-sized C programs. The algorithm for learning disjunctive strategies [13] took 54 hours although only 4 Java programs were used as training data.

In this paper, we present a scalable learning algorithm for data-driven program analysis. The key feature of our algorithm is that it does not require running the static analyzer multiple times over the codebases. To this end, we use an oracle that quantifies the relative importance of program components (e.g., variables) to transform the blackbox optimization problem into a whitebox one that is much easier to solve than the original problem. The oracle can be easily obtained by running the least and most precise analyses over the codebases only once, thereby allowing large codebases to be used as training data.

The experimental results show that our learning algorithm is much faster than the existing algorithm while producing cost-effective strategies. We implemented our approach in the Sparrow static analyzer[1] and applied it to control its strategies for flow-sensitivity and widening-with-thresholds. We used a large codebase, comprising 140 open-source benchmarks (a total of 2.1 MLoC), for evaluation. For widening with thresholds, the learned strategy achieves 88% of 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 1.8x. Our learning algorithm is able to achieve this performance 15.6 times faster than the existing learning algorithm based on Bayesian optimization. We also obtained similar results for flow-sensitivity.

Contributions. This paper, which is an extension of [3], makes the following contributions.

- We present an oracle-guided learning algorithm that is significantly faster than the existing Bayesian optimization approach.
- We prove the effectiveness of our method in a realistic setting. We evaluate the technique with two instance analyses: flow-sensitivity and widening-with-thresholds. We used a large codebase of 140 open-source programs (a total of 2.1 MLoC), which was not possible before due to the cost of learning algorithms.

This paper extends the previous conference version [3] as follows:

- It generally describes our learning algorithm and the idea for obtaining oracles efficiently (Section 3.4). In [3], the overall approach was specific to the problem of finding widening thresholds.
- We show that our approach is also applicable to controlling flow-sensitivity (Section 4.1). In particular, we compare the result with that in prior work [21] used for flow-sensitivity (Section 5.6).

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 strategy for each instance analysis. 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 [21]. Let $\mathbb{P}$ be a set of programs and $P \in \mathbb{P}$ 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, $\mathbb{J}_P$ represents the set of program variables and the analysis applies flow-sensitivity only to a selected subset of $\mathbb{J}_P$. In another instance analysis which applies the technique of widening thresholds, $\mathbb{J}_P$ 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 latter. According to our experience, this assumption usually holds in program analyses for C.

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

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

[1] http://github.com/ropas/sparrow

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 [21] but uses a new learning algorithm that is much more efficient than the Bayesian optimization approach in [21].

In Section 3.1, we summarize the definition of the adaptation strategy in [21], 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 [21], 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. In this paper, we assume that k is fixed and $\mathbb{R}$ denotes real numbers between -1 and 1, i.e., $\mathbb{R} = [-1, 1]$. The strategy is parameterized by $\mathbf{w} \in \mathbb{R}^n$, a vector of the real numbers. Hence, 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:

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

Then, the strategy $\mathcal{S}_{\mathbf{w}}$ works in two steps. Let us explain how it works with the following example in the flow-sensitive analysis:

```
1 int x=inputs(); int y=0; int z=15;
2 y = z + 5;
3 assert(y >= 20);
```

Suppose that we use the 4-dimensional feature vectors of program variables `x`, `y`, `z` as follows:

$$\mathbf{f}_P(x) = \langle 1, 1, 1, 1 \rangle$$
$$\mathbf{f}_P(y) = \langle 0, 1, 1, 1 \rangle$$
$$\mathbf{f}_P(z) = \langle 0, 1, 0, 0 \rangle$$

1. We first compute the scores of variables. The score of variable x is computed by a linear combination of its feature vector and the parameter $\mathbf{w}$:

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

   The parameter $\mathbf{w}$ has the same dimension as the feature vector. Suppose that the parameter $\mathbf{w} \in \mathbb{R}^4$ is given as follows:

$$\mathbf{w} = \langle -0.7, 0.3, 0.4, 0.2 \rangle$$

   The scores of variables `x`, `y`, `z` in the code are calculated as follows:

$$score_P^{\mathbf{w}}(x) = \langle 1, 1, 1, 1 \rangle \cdot \langle -0.7, 0.3, 0.4, 0.2 \rangle = 0.2$$
$$score_P^{\mathbf{w}}(y) = \langle 0, 1, 1, 1 \rangle \cdot \langle -0.7, 0.3, 0.4, 0.2 \rangle = 0.9$$
$$score_P^{\mathbf{w}}(z) = \langle 0, 1, 0, 0 \rangle \cdot \langle -0.7, 0.3, 0.4, 0.2 \rangle = 0.3$$

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

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

2. We select the top-k variables based on the their scores. Our strategy selects top-k variables with highest scores:

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

   For instance, when $k = 1$, we choose the variable `y`. In this case, we can prove the assertion even though only variable `y` is analyzed with flow-sensitivity.

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:

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

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 [21], 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 $\mathbf{w}$ from $\mathbb{R}^n$ using probabilistic model $\mathcal{M}$
3: $s \leftarrow obj(\mathbf{w})$
4: update the model $\mathcal{M}$ with $(\mathbf{w}, s)$
5: **until** timeout
6: **return** best $\mathbf{w}$ 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 [22]). 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 [21], 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 50 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 running the most precise analysis $F(P, \mathbb{J}_P)$ and the least one $F(P, \emptyset)$ over each program P 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 $j \in \mathbb{J}_P$, 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.

The high-level idea for obtaining an oracle $\mathcal{O}_P$ in program P is to compare the results at the fixed points obtained from the most precise ($F(P, \mathbb{J}_P)$) and the least precise ($F(P, \emptyset)$) analyses, and then find out their differences in terms of the program components ($\mathbb{J}_P$). Intuitively, if there is no difference, applying the precise but costly technique to such program component j is definitely unnecessary. For instance, in a flow-sensitive analysis, where $j \in \mathbb{J}_P$ represents a program variable in P, we identify the program variables whose analysis results improve with flow-sensitivity. In Section 4, we show how to obtain such an oracle concretely for two instance analyses.

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: $\mathbf{w} = \mathbf{w} - \alpha \cdot \nabla E(\mathbf{w})$
4: **until** convergence
5: **return** $\mathbf{w}$

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 running codebase with the most and least precise analyses 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. Instance Analyses

In this section, we explain how to employ the oracle-guided method to learn an effective strategy from a codebase for two instance analyses: flow-sensitive analysis and widening-with-thresholds.

4.1. Learning a Strategy for Flow-Sensitive Analysis

First instance analysis is a partial flow-sensitive analysis, where the notations and concepts in this subsection follow [21]. By using an illustrative code example, we also present how to obtain the oracle for a partial flow-sensitive analysis.

Sparse Flow-Sensitive Analysis. 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 program points to abstract states:

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

An abstract state $s \in \mathbb{S}$ is a map from abstract locations (i.e., program variables) to abstract values:

$$\mathbb{S} = \mathbb{L} \to \mathbb{V}.$$

We assume that for each program point c, the analysis has a transfer function $f_c : \mathbb{S} \to \mathbb{S}$ that defines the abstract meaning of the command at c.

The flow-sensitive analysis is defined on top of the sparse analysis framework [18]. We write $\mathsf{D}(c) \subseteq \mathbb{L}$ and $\mathsf{U}(c) \subseteq \mathbb{L}$ for the definition and use sets at program point $c \in \mathbb{C}$. With $\mathsf{D}(c)$ and $\mathsf{U}(c)$, the data-dependency relation $(\rightsquigarrow) \subseteq \mathbb{C} \times \mathbb{L} \times \mathbb{C}$ is defined as follows [18]:

$$\begin{aligned} c_0 \overset{l}{\rightsquigarrow} c_n \quad = \quad & \exists [c_0, c_1, \ldots, c_n] \in \mathsf{Paths}, l \in \mathbb{L} \\ & l \in \mathsf{D}(c_0) \cap \mathsf{U}(c_n) \wedge \forall 0 < i < n. \, l \notin \mathsf{D}(c_i) \end{aligned}$$

Here, $c_0 \overset{l}{\rightsquigarrow} c_n$ means that the location l defined at program point c_0 is used in c_n and l is not redefined at any of the intermediate program points c_i. With the data dependence relation, the sparse analysis is defined as follows:

$$F(X) = \lambda c. \; f_c\big(\lambda l. \bigsqcup_{c_0 \overset{l}{\rightsquigarrow} c} X(c_0)(l)\big).$$

We say this analysis is fully flow-sensitive because it constructs data dependencies for every abstract location and tracks all the dependencies accurately.

We extend this sparse-analysis framework to support selective flow-sensitivity, where an analysis is allowed to track data dependencies only for a subset $L \subseteq \mathbb{L}$ of abstract locations. For the remaining locations (i.e., $\mathbb{L} \setminus L$), we apply flow-insensitivity by using results from a quick flow-insensitive pre-analysis [18]. Let $s_i \in \mathbb{S}$ be the results of the flow-insensitive pre-analysis. The partial data-dependency with respect to L is defined as follows:

$$\begin{aligned} c_0 \overset{l}{\rightsquigarrow}_L c_n \quad = \quad & \exists [c_0, c_1, \ldots, c_n] \in \mathsf{Paths}, l \in L. \\ & l \in \mathsf{D}(c_0) \cap \mathsf{U}(c_n) \wedge \forall 0 < i < n. \, l \notin \mathsf{D}(c_i) \end{aligned}$$

That is, the relation $c_0 \overset{l}{\rightsquigarrow}_L c_n$ holds only when the location l is included in the set L. With this notion of partial data dependency, we next define the semantics function:

$$F_L(X) = \lambda c. \; f_c(s')$$
$$\text{where } s'(l) = \begin{cases} X(c)(l) & (l \notin L) \\ \bigsqcup_{c_0 \overset{l}{\rightsquigarrow}_L c} X(c_0)(l) & \text{otherwise} \end{cases}$$

This definition means that the abstract state at program point c is the flow-insensitive result $s_I(l)$ if the location l is not in L. Otherwise, we follow the original treatment

of the sparse analysis. With this formulation, we compute $\mathsf{lfp}_{X_0} F_L$, where the initial $X_0 \in \mathbb{D}$ is

$$X_0(c)(l) = \begin{cases} s_I(l) & l \notin L \\ \bot & \text{otherwise} \end{cases}$$

Note that L controls the degree of flow-sensitivity. For instance, when $L = \mathbb{L}$, the analysis performs an ordinary flow-sensitive sparse analysis. On the other hand, when $L = \emptyset$, the analysis is identical to a flow-insensitive analysis.

Features. To use the learning algorithm, we need a set of features for abstract locations (e.g., program variables, structure fields and allocation sites) in the program. To do this, we reused the same 45 features which have been already designed in [21].

Oracle. For the instance analysis, we define the oracle as follows:

$$\mathcal{O}_P^F : \mathbb{L}_P \to \mathbb{R}$$

where $\mathbb{L}_P$ is the set of locations (e.g., program variables) in the program P. Now, $\mathcal{O}_P^F$ maps the program variables into their relative importance when they are treated flow-sensitively.

To build an oracle in flow-sensitive analysis, our heuristic works in three steps. Consider the following program:

```
1 int x=inputs(); int y=0; int z=15;
2 y = z + 5;
3 assert(y >= 20);
```

1. We run the program twice, one with flow-sensitive analysis and the other with flow-insensitive analysis.
2. We find out the interval values of variables from both analysis results. At line 3, the interval values of each variable for the two analyses are as follows:

   $$\begin{array}{c} \textit{Sensitive} : x \to [-\infty, \infty], y \to [20, 20], z \to [15, 15] \\ \textit{Insensitive} : x \to [-\infty, \infty], y \to [0, 20], z \to [15, 15] \end{array}$$

   That is, the interval value of variable y is [20, 20] in flow-sensitive analysis while it is [0, 20] in flow-insensitive one.
3. We compare the difference of each variable obtained from the two analyses as follows:

   $$[x \to 0, y \to 1, z \to 0]$$

   For simplicity, we assume that the result is binary; if the interval value of each variable between the two analyses has difference, the variable (e.g., y) is important, marked as 1. if not, the variable (e.g., x, z) is not important. Then, we obtain the oracle $\mathcal{O}_P^F : [x \to 0, y \to 1, z \to 0]$. We repeat this process over the entire codebase and generate a set of oracles for each program in the codebase.

4.2. Learning a Strategy for Widening Thresholds

We define an interval analysis that uses widening with thresholds. Then, we present the features and oracle that we used for the interval analysis, respectively.

Interval Analysis with Widening Thresholds. We follow the notations defined in Section 4.1. Hence, the abstract semantic function of the analysis with widening thresholds is defined as follows:

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

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).

$$\begin{array}{l} [l_1, u_1] \triangledown [l_2, u_2] = \\ \qquad [(l_2 < l_1 ? -\infty : l_1), (u_1 < u_2 ? +\infty : u_1)] \end{array} \quad (3)$$

Note that this widening operator is very hasty and immediately replaces unstable bounds by ∞. The unstable bounds denote the lower and upper bounds of intervals that have not reached the fixed point yet.

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:

$$\begin{array}{l} [l_1, u_1] \triangledown_T [l_2, u_2] = \\ \quad [(l_2 < l_1 ? \mathit{glb}(T, l_2) : l_1), (u_1 < u_2 ? \mathit{lub}(T, u_2) : u_1)] \end{array} \quad (4)$$

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

$$\begin{array}{rcl} \mathit{glb}(T, i) & = & \max\{n \in T \mid n \leq i\} \\ \mathit{lub}(T, i) & = & \min\{n \in T \mid n \geq i\} \end{array}$$

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 2 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 $\mathbf{w}$.

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 $n > 1000$

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.

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

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

where $\mathbb{Z}_P$ is the set of integer constants that appear in the program P and $\mathbb{R} = [-1, 1]$. That is, $\mathcal{O}_P^T$ maps integer constants in the program into their relative importance when they are used for widening thresholds.

We illustrate the simple heuristic for building such an oracle. Our heuristic works in the three steps when considering the following program P:

```
1 char *text="abcd"; int i=0;
2 while (text[i] != NULL) {
3   i = i + 1;
4   assert(i <= 4);
5 }
```

Note that when an integer 4 is used as a widening threshold, we can prove that the assertion holds at line 4. This is possible because we can convert the loop condition "text[i] != NULL" into an equivalent one "i != 4" (where 4 is the NULL position of text). Then, when an integer 4 is used as a threshold, the widening operation at the loop head produces the interval [0,4], instead of [0,$+\infty$], for the value of i. So, the loop condition "i != 4" can narrow down the value of i to $[0, 3]$ and therefore we can prove the assertion holds.

1. We analyze the program P twice, in the most and least precise settings. The most precise analysis uses all integer constants in P as widening thresholds, including the interval values of the program's variables and the sizes and offsets of static arrays and so on. We can compute them from a cheap flow-insensitive pre-analysis. That is, for program P, the integer constants, including 0, 1, 4 and 5 (size of array `text`), are used as widening thresholds in the most precise analysis. On the other hands, the least precise analysis does not use any integer constant in P as a widening-threshold. That is, it performs the interval analysis with the basic widening operator in Equation (3).
2. During the two analyses, we count how many times each integer constant is used as lower and upper bounds of intervals at the fixed point of loop head

(line 2), which we use as an estimate of the importance of integer constants. As a result, because the fixed point values of variable `i` at loop head in the most and least precise analyses are $[0, 4]$ and $[0, +\infty]$, respectively, the importance of each constant in the two analyses is counted as follows:

$$\begin{array}{rcl} \textit{most precise} & : & [0 \to 1, 4 \to 1] \\ \textit{least precise} & : & [0 \to 1] \end{array}$$

That is, the constant 4 is used at the fixed point in the most analysis but not in the least precise one.

3. We calculate the difference the results obtained from the two analyses as follows:

$$[4 \to 1, 0 \to 0]$$

Finally, we normalize the values to obtain the oracle $\mathcal{O}_P^T : [4 \to 1, 0 \to 0]$. We repeat this process over the entire codebase and generate a set of oracles.

4.3. Learning a Strategy for Combined Analysis

Our approach can also learn a strategy for combined analysis that controls both flow-sensitivity and widening with thresholds simultaneously. In the combined analysis, the set $\mathbb{J}_P$ in Section 2 represents all program variables and integer constants appeared in program P. The job of the strategy is to choose a subset of $\mathbb{J}_P$ that high precision techniques will be applied. For the combined one, we reused all features designed in Section 4.1 and 4.2.

Oracle. For building an oracle in the combined analysis, we follow the similar steps just as we obtain the oracle for a particular analysis in Section 4.1 and 4.2, respectively.

We run the codebase twice, with both the most precise and the least precise analysis. In the combined analysis, the former denotes flow-sensitive analysis that uses all integer constants in P as widening thresholds; latter is flow-insensitive analysis with the basic widening operator in (3). Using each heuristic explained in Section 4.1 and 4.2, we can easily obtain both oracles for the combined analysis: $\mathcal{O}_P^F$ and $\mathcal{O}_P^T$.

5. Experiments

In this section, we empirically evaluate our approach with an interval analyzer for C programming language. We conducted the experiments to answer the following research questions:

1. **Effectiveness of learned strategy**: How much is our analyzer with learned strategy better than the baseline analyzers for flow-sensitive analysis (Section 5.2), widening with thresholds (Section 5.3), and combined analysis (Section 5.4), respectively?
2. **Impact of the number of training programs**: Are the large codebases necessary to learn good strategies? How does the number of training programs affect the qualities of the learned strategies? (Section 5.5)
3. **Efficacy of the learning algorithm**: How much is our learning algorithm better than the existing Bayesian optimization approach? (Section 5.6)
4. **Important Features**: What are the most important features identified by the learning algorithm? (Section 5.7)

5.1. Setting

We implemented our approach in Sparrow, a static buffer-overflow analyzer for real-world C programs [19]. The analysis is based on the interval abstract domain and performs a flow-sensitive and selectively context-sensitive analysis [20]. Along the interval analysis, it also simultaneously performs a flow-sensitive pointer analysis to handle indirect assignments and function pointers in C. The analyzer also takes as arguments the program parts to which high precision technique will be applied. For instance, for analysis with widening thresholds, the analyzer takes 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 140 open-source C programs (0.4−93KLoC) from GNU and Linux packages. The list of programs we used is available in Table 6. For flow-sensitive analysis, we randomly divided the 140 benchmark programs into 100 training programs and 40 testing programs. A strategy for choosing program variables is learned from the 100 training programs, and tested on the remaining 40 programs. We iterated this process for five times. In experiments, we set k to roughly 15%, which means that the strategy approximately chooses the top 15% program variables to use for flow-sensitive analysis. In fact, we have tried to set k to different values (e.g., 5%, 10% and 20%). However, we found that the most balanced results were produced when k was 15%.

For analysis with widening thresholds, we used only 80 programs, a subset of total benchmarks because for some programs, the technique of widening thresholds did not improve the analysis precision at all. Table 7 shows the list of programs we used. We also randomly divided the 80 benchmark programs into 50 training programs and 30 testing programs. We repeated this process for five times. For this instance analysis, 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. Table 2 and 3 show the result of each trial for flow-sensitive analysis and widening-with-thresholds, respectively.

Finally, for the combined analysis in Section 4.3, we reused the settings fixed for each instance analysis. That is, our strategy simultaneously chooses the top 15% program variables and top 30 integer constants. For the evaluation, we also reused five training sets prepared for analysis with widening thresholds. In Table 4, we report the effectiveness of our strategy learned for combined analysis.

Table 2: Effectiveness of our method for flow-sensitive analysis. Quality = (d-a)/(c-a), Cost = e/b

Trial	Fi	Fs	Foracle	
	prove	prove	prove	quality
1	17,120	19,410	18,831	74.7 %
2	19,846	22,625	22,030	78.6 %
3	19,219	21,913	21,310	77.6 %
4	22,473	25,067	24,565	80.6 %
5	20,216	22,886	22,410	82.2 %
TOTAL	98,874	111,901	109,146	**78.9 %**

(a) Training Result

Trial	Fi		Fs			Foracle			
	prove (a)	sec (b)	prove (c)	sec	cost	prove (d)	sec (e)	quality	cost
1	11,815	142	13,294	861	6.1 x	13,026	295	81.9 %	2.1 x
2	9,089	98	10,079	472	4.8 x	9,907	215	82.6 %	2.2 x
3	9,716	145	10,791	933	6.4 x	10,616	286	83.7 %	2.0 x
4	6,462	259	7,637	2,313	8.9 x	7,324	569	73.4 %	2.2 x
5	8,719	254	9,818	2,643	10.4 x	9,559	485	76.4 %	1.9 x
TOTAL	45,801	898	51,619	7,222	**8.0 x**	50,432	1,850	**79.6 %**	**2.1 x**

(b) Testing Result

5.2. Effectiveness for Flow-Sensitive Analysis

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

- Fi : Sparrow that do flow-insensitive analysis. That is, it performs the least precise but, cheap analysis.
- Fs : Sparrow that do flow-sensitive analysis. That is, it performs the most precise but, costly analysis.
- Foracle : Sparrow with the learned strategy. That is, the argument (e.g., program variables) of our analyzer is given by the strategy learned from the 100 programs via our oracle-guided learning algorithm.

Training. For flow-sensitive analysis, Table 2(a) shows the effectiveness of the learned strategy in the training phases for the five trials. As a result, the least precise analyzer (Fi) proved the absence of buffer-overruns over 98,874 queries. On the other hand, the most precise analyzer (Fs) proved 111,901 queries. For 100 training programs, our learning algorithm was able to find a strategy that can prove 78.9% of the Fs-only provable queries.

Testing. Table 2(b) shows the results on the 40 testing programs. In total, Fi proved the 45,801 queries, while Fs proved 51,619 queries. Our analysis with the learned strategy (Foracle) proved 50,432 queries, achieving 79.6% of the precision of Fs. In doing so, Foracle increases the analysis time of Fi only 2.1x, while Fs increases the cost by 8.0x. The time for extracting features is negligible compared to the total analysis time. For example, it took less than a second to extract features from "rnv-1.7.10.c", the biggest program of the benchmarks.

Table 2 also shows that testing qualities were better than training qualities for three trials. For instance, training quality in trial 1 was 74.7% while testing quality was 81.9%. The unusual phenomenon is due to some outlier programs (e.g, `combine-0.3.3` and `chrony-1.29` in Table 6). That is, our learning algorithm fails to find the strategy that can prove Fs-only provable queries in such programs. Hence, when the outlier programs are included in training set, the quality on unseen testing programs can be even better than training quality.

5.3. Effectiveness for Analysis with Widening Threshold

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

- NoThld : Sparrow without widening thresholds. That is, it performs the interval analysis with the basic widening operator in (3).
- FullThld : Sparrow that uses all the integer constants in the program as widening thresholds. The thresholds set includes constant integers in the program, the sizes and offsets of static arrays and the lengths of constant strings, and so on.
- OracleThld : Sparrow whose threshold strategy is learned from the codebase. That is, the threshold argument of the analyzer is given by the strategy learned from the 50 programs.

Training. For the instance analysis, Table 3(a) shows the training performance with 50 programs. For the five trials, NoThld proved 66,414 buffer-overrun queries. On the

Table 3: Effectiveness of our method for analysis with widening thresholds.

Trial	NoThld	FullThld	OracleThld	
	prove	prove	prove	quality
1	15,281	16,681	16,488	86.2 %
2	13,914	15,046	14,819	79.9 %
3	11,790	12,859	12,693	84.5 %
4	13,049	14,341	14,173	87.0 %
5	12,380	13,913	13,685	85.1 %
TOTAL	66,414	72,840	71,858	**84.7 %**

(a) Training Result

Trial	NoThld		FullThld			OracleThld			
	prove	sec	prove	sec	cost	prove	sec	quality	cost
1	5,801	2,119	6,462	8,426	4.0 x	6,402	3,911	90.9 %	1.8 x
2	7,168	754	8,097	3,420	4.5 x	8,034	1,204	93.2 %	1.6 x
3	9,292	2,753	10,284	11,939	4.3 x	10,153	5,179	86.8 %	1.9 x
4	8,033	845	8,802	3,788	4.5 x	8,665	1,384	82.2 %	1.6 x
5	8,702	242	9,230	1,147	4.7 x	9,178	354	90.2 %	1.5 x
TOTAL	38,996	6,713	42,875	28,720	**4.3 x**	42,432	12,032	**88.6 %**	**1.8 x**

(b) Testing Result

other hand, FullThld proved 72,840 queries. For the training programs, our learning algorithm was able to find a strategy that can prove 84.7% of the FullThld-only provable queries.

Testing. Table 3(b) also shows the results on the 30 testing programs. In total, NoThld proved the 38,966 queries, while FullThld proved 42,875 queries. Our analysis with the learned strategy (OracleThld) proved 42,432 queries, achieving 88.6% of the precision of FullThld. In doing so, OracleThld increases the analysis time of NoThld only 1.8x, while FullThld increases the cost by 4.3x.

5.4. Effectiveness for Combined Analysis

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

- Fi+NoThld : Flow-insensitive analysis with the basic widening operator in (3). That is, it performs the least precise combined analysis.
- Fs+FullThld : Flow-sensitive analysis which uses all the integer constants in the program as widening thresholds. That is, it performs the most precise combined analysis.
- Foracle+OracleThld : Analysis with learned flow-sensitivity and widening-with-thresholds. That is, two arguments of the analysis (e.g., program variables and integer constants) are given by the strategy learned from 50 programs.

Training. Table 4(a) shows learning an effective strategy for combined analysis is much more challenging than learning the strategy for an instance analysis. In total trials, the least precise analyzer (Fi+NoThld) proved 57,811 buffer-overrun queries while the most precise one (Fs+FullThld) succeeded in proving 72,840 queries. For 50 training programs, our learning algorithm was able to find a strategy that can prove 61.6% of the Fs+FullThld-only provable queries. The training result is not satisfactory when compared with each result for a particular analysis; Foracle and OracleThld can prove 78.9% of the Fs-only provable queries and 84.7% the FullThld-only provable queries, respectively.

Testing. Table 4(b) shows that our analyzer with learned strategy (Foracle+OracleThld) can significantly reduce the analysis time when compared with the time of Fs+FullThld. For five trials, Fi+NoThld proved the 38,989 queries, while Fs+FullThld proved 42,875 queries. For 30 training programs, our analyzer proved 39,762 queries, achieving 65.0% of the precision of Fs+FullThld. To achieve this, our analyzer increases the analysis time of Fi + NoThld only 3.1x, while Fs+FullThld increases the cost by 32.3x.

Note that by controlling the k value, we are able to adjust the balance between analysis precision and cost. That is, if our aim is to increase the precision, we can use a higher k value. For instance, in the first trial of Table 4(b), we used relatively smaller k values: $k = 15\%$ and $k = 30$ for flow-sensitivity and widening-thresholds, respectively. The result is an analysis that is fast but not very precise; it proves 64.7% of the Fs+FullThld-only provable queries while increasing the cost of Fi+NoThld

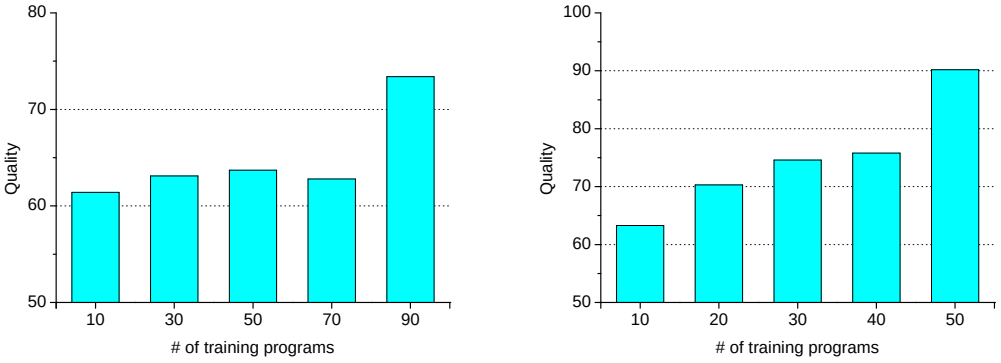
Table 4: Effectiveness of our method for combined analysis.

Trial	Fi + NoThld	Fs + FullThld	Foracle + OracleThld	
	prove	prove	prove	quality
1	13,342	16,681	15,483	64.1 %
2	12,288	15,046	14,059	64.2 %
3	10,108	12,859	12,077	71.6 %
4	11,379	14,341	13,102	58.2 %
5	10,694	13,913	12,345	51.3 %
TOTAL	57,811	72,840	67,066	**61.6 %**

(a) Training Result

Trial	Fi + NoThld		Fs + FullThld			Foracle + OracleThld			
	prove	sec	prove	sec	cost	prove	sec	quality	cost
1	5,018	295	6,462	8,421	28.5 x	5,952	783	64.7 %	2.7 x
2	6,072	78	8,097	3,683	47.2 x	7,126	374	52.0 %	4.8 x
3	8,252	336	10,284	11,958	35.6 x	9,530	972	62.9 %	2.9 x
4	6,981	125	8,802	3,788	30.3 x	8,261	445	70.3 %	3.6 x
5	7,666	63	9,230	1,150	18.3 x	8,893	168	78.5 %	2.7 x
TOTAL	38,989	897	42,875	29,000	**32.3 x**	39,762	2,742	**65.0 %**	**3.1 x**

(b) Testing Result



(a) flow-sensitive (b) widening with thresholds

Figure 1: Learning quality on testing programs by the number of training programs for two instance analyses

by 2.7x. To improve the precision, we can use higher k values. For example, when we set k to 30% and 50 for flow-sensitivity and widening thresholds, respectively, the combined analysis becomes to 80.8% while increasing the cost by 4.2x.

5.5. Impact of the number of training programs

In this subsection, we check our motivation of using large codebases as training programs. To summarize, the performance of the learned strategies overall increases as more training programs are used (Figure 1).

For flow-sensitive analysis, we randomly selected 10, 30, 50, 70, and 90 programs from the 100 training programs, learned five strategies from each subset of the training programs, and then evaluated the learned strategies on the 40 testing programs. Figure 1(a) shows the results. With 90 training programs, the learned strategy could prove 73% of the Fs-only provable queries but the number reduces to 61% when 10 programs are used in the training phase.

For widening with thresholds, we generated five strategies with randomly chosen 10, 20, 30, 40, 50 training programs. The evaluation results on the testing programs are given in Figure 1(b). It shows that the strategy learned with 50 programs can prove 90% of the FullThld-only provable queries. However, the strategy learned with 10 programs managed to prove 63%.

These results support our claim: learning a strategy with large codebases is essential and therefore a scalable learning algorithm is required to make learning feasible.

Table 5: Learning time comparison with the Bayesian optimization for two instance analyses. For Bayesian optimization, we set timeout to 400,000 seconds.

Trial	Quality	ORACLE (sec)	Bayesian Opt (sec)	Speed Up
1	74%	3,849	22,257	5.8x
2	78%	4,238	33,889	8.0x
3	77%	3,777	8,529	2.3x
4	80%	2,397	33,100	13.8x
5	82%	2,067	37,044	17.9x
TOTAL	78%	16,328	134,819	**8.3x**

(a) Flow-Sensitive Analysis

Trial	Quality	ORACLETHLD (sec)	Bayesian Opt (sec)	Speed Up
1	86%	7,598	241,035	31.7x
2	79%	16,181	114,882	7.1x
3	84%	3,024	39,142	12.9x
4	87%	15,732	Timeout (400,000)	$>$ 25.4x
5	85%	19,255	167,367	8.7x
TOTAL	84%	61,790	962,426	**15.6x**

(b) Widening-with-Thresholds

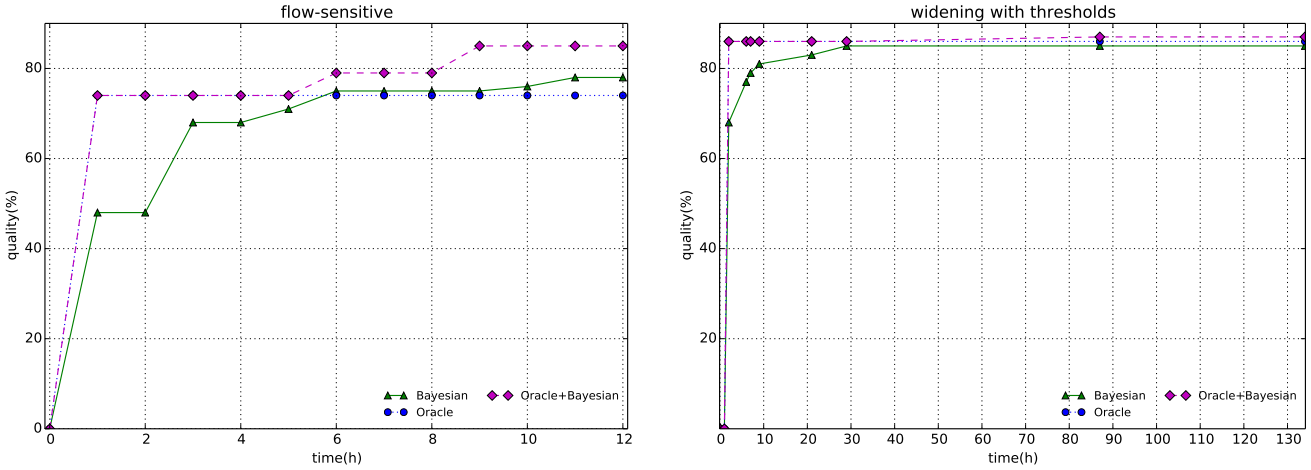


Figure 2: Comparison between the oracle-guided algorithm and Bayesian optimization algorithm

5.6. Efficacy of the learning algorithm

We have implemented the previous learning algorithm based on Bayesian optimization [21]. Then, given a target quality which the learnt strategy should achieve on each training set, we compared its learning time with that of our oracle-guided learning algorithm for two instance analyses.

Table 5 shows that our learning algorithm significantly reduces the learning burden. First, for flow-sensitive analysis, our approach took on average 3,266 seconds to find a strategy of the average quality 78.0% for the five trials (Table 5(a)). On the other hand, the Bayesian optimization approach took on average 26,964 seconds to find a strategy of the same quality on training sets. Second, for widening-with-thresholds, our approach took on average 12,358 seconds to find a strategy of the average quality 84.0% while the Bayesian optimization approach took on average 192,485 seconds to find a strategy of the same one (Table 5(b)). That is, our learning algorithm is able to find the same strategy 8.3 and 15.6 times faster than the existing algorithm for flow-sensitivity and widening-with-thresholds, respectively.

The Bayesian optimization approach did not also work well with a limited time budget. When we allowed the Bayesian optimization approach to use the same time budget as ours, we compared the quality of the strategy learned from the existing approach with that of ours on training and testing sets in Table 2 and 3.

First, for flow-sensitive analysis, the existing approach ended up with a strategy of the average quality 47.1% in training phases for five trials. Note that our algorithm achieves the quality 78.9% in the same amount of time. For the testing sets, the analyzer with a strategy learnt

from the existing approach proves about only 52.8% of Fs-only queries while our analyzer proves about 79.6% of Fs-only queries. Second, for widening-with-thresholds, we also obtained similar results. Given the same time budgets for learning, the existing approach ended up with a strategy of the average quality 62.4% and 54.6% in training and testing phases, respectively. However, our approach learns an effective strategy of the average quality 84.7% and 88.6%, respectively, in the same amount of time.

Given more resources (e.g., times) for both learning algorithms, we additionally evaluate which one has better performance for the first training sets of two analyses in Table 2 and 3. To do it, we gave both algorithms 44,514 seconds and 482,070 seconds, doubling the Bayesian learning time in Table 5. Figure 2 shows the change of the best quality found by each learning algorithm over time. For flow-sensitive analysis, it took Bayesian optimization algorithm about 6 hours to beat our algorithm. After 11 hours, Bayesian algorithm found the best parameter having 78% quality while the best quality of our algorithm ended up 74%. For analysis with widening thresholds, despite being given 134 hours, Bayesian algorithm failed to find a better parameter.

One interesting point is that combining both algorithms can produce much better results. To achieve this, the process consists of two steps. First, we run the oracle-guided algorithm once to obtain a good initial parameter. Second, starting with this parameter (not random parameter), we run Bayesian optimization algorithm until a fixed learning budget runs out. Figure 2 also shows the combined algorithm succeeds in finding better parameters having 85% and 87% for two analyses, respectively.

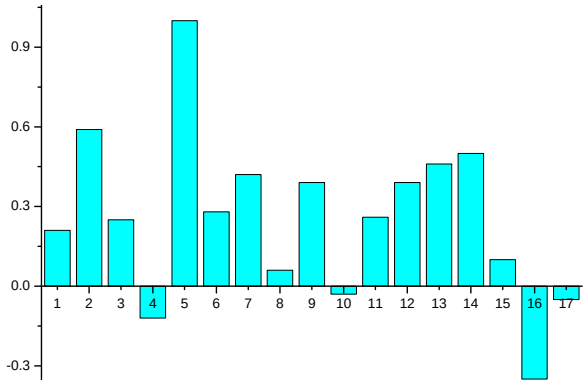


Figure 3: Relative importance among features for widening thresholds

5.7. Important Features

In our approach, the learned parameter **w** in Table 3 indicates the relative importance of the features in Table 1. To identify the important features for widening thresholds, we averaged the parameters obtained from each training set in Table 3.

Figure 3 shows the relative feature importance identified by the learning algorithm. During the five trials, the feature 5 (most frequently appeared numbers in the program) was always the highest ranked feature. Feature 2 (the size of a static array -1), 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:

```
1  char *text="abcd"; int i=0;
2  while (text[i] != NULL) {
3    i = i + 1;
4    assert(i <= 4);
5  }
```

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 2, 5, 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.

Impact of Combining Diverse Features. The combined use of the features in Section 4 is crucial for learning an effective strategy from a codebase. That is, a simple strategy having only a single feature failed to build a cost-effective static analyzer. We evaluated the performance of the simple strategies for two instance analyses. A single feature we used for flow-sensitive analysis is the feature "program variables indirectly used in realloc" in [21]. For analysis with widening thresholds, we used the feature 5 (i.e. the most frequently appeared numbers in the program) in Table 1. These features were chosen because they are the highest ranked features in our strategies learned for each analysis.

The experimental results show that the performance of the simple strategy is inferior to the one of our learned strategy. The simple one for flow-sensitive analysis was able to prove 30.3% of Fs-only provable queries for 5 training sets in Table 2. Note that ours proved 78.9% of Fs only provable queries. For another analysis, the simple one having only feature 5 was able prove 77.1% of FullThld only provable queries for total trials in Table 3 while ours achieved 84.5% of the precision of FullThld.

6. Related Work

Data-Driven Program Analysis. Recently, various techniques for data-driven program analysis have been proposed [21, 3, 11, 12, 4, 13]. Towards building a cost-effective analyzer, these works share a common high-level methodology which automatically learns a strategy for the specific analysis from codebase. For instance, [11] presented a method for automatically learning a variable-clustering strategy for the Octagon analysis. In [12], a new technique for learning a strategy which selectively employs unsoundness for the taint and interval analyses was proposed. For context-sensitive points-to analysis, [13] proposed a method for automatically learning boolean formulas that decide when and how much to employ context-sensitivity.

However, reducing the learning cost is not the main concern of these approaches, which is the main goal of this paper. In particular, our work is motivated by the result of [21], 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 precise analysis on all training programs repeatedly until the learnt strategy meets a target quality. As we mentioned in Sec. 5.6, it takes too much time to get an acceptably good strategy over the large codebase (a total of 2.1 MLoC). 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 the most and least precise analyses per training program only once, our learning algorithm radically reduced time cost than the existing method.

Parametric Program Analysis. A number of techniques for parametric program analysis have been proposed to develop a strategy for finding good abstractions in target programs [20, 28, 23, 27, 1, 5, 6, 15, 9]. To achieve the goal, techniques such as pre-analyses [20, 23] and counterexample-guided abstraction refinement (CEGAR) [28, 27] have been used. However, these works focus on manually developing a fixed strategy while our approach aims to automatically find an adaptive strategy from codebase.

For the analysis with widening thresholds, existing techniques use a fixed strategy for choosing the threshold set [1, 5, 6, 15]; all the integer constants that appear in conditional statements are used for the candidate of thresholds. In [9], 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 [9, 15]. 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.

7. Conclusion

In this paper, we proposed a new learning algorithm for data-driven program analysis. Our algorithm overcomes the scalability limitation of the existing algorithm and allows large codebases to be used as training data. In the presence of a large codebase, comprising of 2.1MLoC, the existing algorithm with Bayesian optimization failed to learn a good strategy in a reasonable amount of time. By contrast, for analysis with widening thresholds, our new learning algorithm is 15 times faster and is able to find a better parameter than the previous method. Our approach is general enough to be used for various types of adaptive static analyses. We applied it to two instance analyses for C programs: flow-sensitivity and widening thresholds. We hope that our technique will make the data-driven program analysis more practical in the real-world.

Acknowledgement. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09. 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).

[1] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2002. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In: The Essence of Computation. Springer, pp. 85–108.

[2] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2003. A Static Analyzer for Large Safety-Critical Software. In: PLDI.

[3] Cha, S., Jeong, S., Oh, H., 2016. Learning a strategy for choosing widening thresholds from a large codebase. In: Asian Symposium on Programming Languages and Systems. Springer, pp. 25–41.

[4] Chae, K., Oh, H., Heo, K., Yang, H., Oct. 2017. Automatically generating features for learning program analysis heuristics for C-like languages. Proc. ACM Program. Lang. 1 (OOPSLA), 101:1–101:25.
URL `http://doi.acm.org/10.1145/3133925`

[5] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Antoine, M., Rival, X., 2009. Why does astrée scale up? Formal Methods in System Design 35 (3), 229–264.
URL `http://dblp.uni-trier.de/db/journals/fmsd/fmsd35.html#CousotCFMMR09`

[6] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2006. Combination of abstractions in the astrée static analyzer. In: Advances in Computer Science-ASIAN 2006. Secure Software and Related Issues. Springer, pp. 272–300.

[7] Grigore, R., Yang, H., 2016. Abstraction refinement guided by a learnt probabilistic model. In: POPL.

[8] Guyer, S., Lin, C., 2003. Client-driven pointer analysis. Static Analysis, 1073–1073.

[9] Halbwachs, N., Proy, Y.-E., Roumanoff, P., 1997. Verification of real-time systems using linear relation analysis. In: FORMAL METHODS IN SYSTEM DESIGN. pp. 157–185.

[10] Heintze, N., Tardieu, O., 2001. Demand-driven pointer analysis. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. PLDI '01.

ACM, New York, NY, USA, pp. 24–34.
URL http://doi.acm.org/10.1145/378795.378802

[11] Heo, K., Oh, H., Yang, H., 2016. Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In: International Static Analysis Symposium. Springer, pp. 237–256.

[12] Heo, K., Oh, H., Yi, K., 2017. Machine-learning-guided selectively unsound static analysis. In: Proceedings of the 39th International Conference on Software Engineering. ICSE '17. IEEE Press, Piscataway, NJ, USA, pp. 519–529.
URL https://doi.org/10.1109/ICSE.2017.54

[13] Jeong, S., Jeon, M., Cha, S., Oh, H., Oct. 2017. Data-driven context-sensitivity for points-to analysis. Proc. ACM Program. Lang. 1 (OOPSLA), 100:1–100:28.
URL http://doi.acm.org/10.1145/3133924

[14] Kastrinis, G., Smaragdakis, Y., 2013. Hybrid context-sensitivity for points-to analysis. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13. ACM, New York, NY, USA, pp. 423–434.
URL http://doi.acm.org/10.1145/2491956.2462191

[15] Kim, S., Heo, K., Oh, H., Yi, K., 2015. Widening with thresholds via binary search. Software: Practice and Experience.

[16] Liang, P., Tripp, O., Naik, M., 2011. Learning minimal abstractions. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '11. ACM, New York, NY, USA, pp. 31–42.
URL http://doi.acm.org/10.1145/1926385.1926391

[17] Miné, A., 2006. The octagon abstract domain. Higher-order and symbolic computation.

[18] Oh, H., Heo, K., Lee, W., Lee, W., Yi, K., 2012. Design and implementation of sparse global analyses for C-like languages. In: PLDI.

[19] Oh, H., Heo, K., Lee, W., Lee, W., Yi, K., 2014. Sparrow. http://ropas.snu.ac.kr/sparrow.

[20] Oh, H., Lee, W., Heo, K., Yang, H., Yi, K., 2014. Selective context-sensitivity guided by impact pre-analysis. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14. ACM, New York, NY, USA, pp. 475–484.
URL http://doi.acm.org/10.1145/2594291.2594318

[21] Oh, H., Yang, H., Yi, K., 2015. Learning a strategy for adapting a program analysis via bayesian optimisation. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2015. ACM, New York, NY, USA, pp. 572–588.
URL http://doi.acm.org/10.1145/2814270.2814309

[22] Rasmussen, C. E., Williams, C. K. I., 2005. Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning). The MIT Press.

[23] Smaragdakis, Y., Kastrinis, G., Balatsouras, G., 2014. Introspective analysis: Context-sensitivity, across the board. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14. ACM, New York, NY, USA, pp. 485–495.
URL http://doi.acm.org/10.1145/2594291.2594320

[24] Sridharan, M., Bodík, R., Jun. 2006. Refinement-based context-sensitive points-to analysis for java. SIGPLAN Not. 41 (6), 387–400.
URL http://doi.acm.org/10.1145/1133255.1134027

[25] Sridharan, M., Gopan, D., Shan, L., Bodík, R., 2005. Demand-driven points-to analysis for java. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '05. ACM, New York, NY, USA, pp. 59–76.
URL http://doi.acm.org/10.1145/1094811.1094817

[26] Tan, T., Li, Y., Xue, J., 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In: International Static Analysis Symposium. Springer, pp. 489–510.

[27] Zhang, X., Mangal, R., Grigore, R., Naik, M., Yang, H., 2014. On abstraction refinement for program analyses in datalog. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14. ACM, New York, NY, USA, pp. 239–248.
URL http://doi.acm.org/10.1145/2594291.2594327

[28] Zhang, X., Naik, M., Yang, H., 2013. Finding optimum abstractions in parametric dataflow analysis. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13. ACM, New York, NY, USA, pp. 365–376.
URL http://doi.acm.org/10.1145/2491956.2462185

Table 6: Benchmark programs for flow-sensitive analysis

Programs	LOC	Programs	LOC
brutefir-1.0f.c	398	lgrind-3.67.c	7,363
wwl-1.3+db.c	474	lacheck-1.26.c	7,385
gosmore-0.0.0.20100711.c	497	httptunnel-3.3.c	7,470
ircmarkers-0.14.c	619	lakai-0.1.c	7,487
consol_calculator.c	1,124	libdebug-0.4.4.c	7,645
rovclock-0.6e.c	1,177	cmigemo-1.2+gh0.20140306.c	7,729
xcircuit-3.7.55.dfsg.c	1,222	mpegdemux-0.1.3.c	7,783
iputils-20121221.c	1,311	barcode-0.96.c	7,901
confget-1.02.c	1,393	apngopt-1.2.c	8,315
dtmfdial-0.2+1.c	1,440	makedepf90-2.8.8.c	8,415
codegroup-19981025.c	1,518	stripcc-0.2.0.c	8,914
id3-0.15.c	1,652	xfpt-0.07.c	9,089
time-1.7.c	1,759	photopc-3.05.c	9,266
polymorph-0.4.0.c	1,764	psmisc-22.20.c	9,624
rexima-1.4.c	1,843	ircd-ircu-2.10.12.10.dfsg1.c	10,206
xinit-1.3.2.c	1,893	man-1.5h1.c	11,059
nlkain-1.3.c	1,927	auto-apt-0.3.23ubuntu0.14.04.1.c	11,110
xchain-1.0.1.c	1,955	glhack-1.2.c	11,237
display-dhammapada-1.0.c	2,007	cjet-0.8.9.c	11,287
authbind-2.1.1.c	2,041	admesh-0.95.c	11,441
unhtml-2.3.9.c	2,057	hspell-1.0.c	11,520
elfrc-0.7.c	2,142	sac-1.9b5.c	11,999
jbofihe-0.38.c	2,182	dict-gcide-0.48.1.c	12,318
delta-2006.08.03.c	2,273	juke-0.7.c	12,518
petris-1.0.1.c	2,411	gzip-spec2000.c	12,980
libixp-0.6∼20121202+hg148.c	2,428	cutils-1.6.c	14,122
mp3rename-0.6.c	2,466	rhash-1.3.1.c	14,352
whichman-2.4.c	2,493	mpage-2.5.6.c	14,827
acpi-1.7.c	2,597	gnuspool-1.7ubuntu1.c	16,665
zmakebas-1.2.c	2,606	ample-0.5.7.c	17,098
forkstat-0.01.04.c	2,710	irmp3-ncurses-0.5.3.1.c	17,195
mp3wrap-0.5.c	2,752	smp-utils-0.97.c	17,520
ncompress-4.2.4.c	2,840	ccache-3.1.9.c	17,536
setbfree-0.7.5.c	2,929	tnef-1.4.6.c	18,172
pgdbf-0.5.0.c	3,135	ecasound2.2-2.7.0.c	18,236
haskell98-tutorial-200006-2.c	3,161	gzip-1.2.4a.c	18,364
mcf-spec2000.c	3,407	unrtf-0.19.3.c	19,019
kcc-2.3.c	3,429	netkit-ftp-0.17.c	19,254
ipip-1.1.9.c	3,605	libchewing-0.3.5.c	19,262
acpi-1.4.c	3,814	jwhois-3.0.1.c	19,375
gif2apng-1.7.c	3,816	archimedes.c	19,552
desproxy-0.1.0∼pre3.c	3,841	tcs-1.c	19,967
magicfilter-1.2.c	3,856	gnuplot-4.6.4.c	20,306
pgpgpg-0.13.c	3,908	phalanx-22+d051004.c	24,099
rsrce-0.2.2.c	3,956	aewan-1.0.01.c	28,667
rinetd-0.62.c	4,123	gnuchess-5.05.c	28,853
unsort-1.1.2.c	4,290	combine-0.3.3.c	29,508
hexdiff-0.0.53.c	4,334	rtai-3.9.1.c	30,739
acorn-fdisk-3.0.6.c	4,450	normalize-audio-0.7.7.c	30,984
checkmp3-1.98.c	4,450	less-382.c	31,623
pmccabe-2.6.c	4,920	tmndec-3.2.0.c	31,890
dvbtune-0.5.ds.c	5,068	fondu-0.0.20060102.c	32,298
bmf-0.9.4.c	5,451	gbsplay-0.0.91.c	34,002
cam-1.05.c	5,459	parser.c	36,178
libbind-6.0.c	5,497	enscript-1.6.5.c	38,787
bottlerocket-0.05b3.c	5,509	libart-lgpl-2.3.21.c	38,815
mixal-1.08.c	5,570	flex-2.5.39.c	39,977
cmdpack-1.03.c	5,575	fwlogwatch-1.2.c	46,601
picocom-1.7.c	5,613	chrony-1.29.c	49,119
xdms-1.3.2.c	5,614	wget-1.9.c	54,219
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	bison-2.4.c	59,955
129.compress.c	6,078	tree-puzzle-5.2.c	62,302
buildtorrent-0.8.c	6,170	icecast-server-1.3.12.c	68,564
e2ps-4.34.c	6,222	dico-2.0.c	69,308
apng2gif-1.5.c	6,522	aalib-1.4p5.c	73,412
isdnutils-3.25+dfsg1.c	6,609	shadow-4.1.5.1.c	85,201
bwm-ng-0.6.c	6,833	skyeye-1.2.5.c	85,905
diffstat-1.58.c	7,077	rnv-1.7.10.c	93,858
Total	**242,128**	**Total**	**1,878,827**

Table 7: Benchmark programs for widening-with-thresholds

Programs	LOC	Programs	LOC
gosmore-0.0.0.20100711.c	497	lakai-0.1.c	7,487
ircmarkers-0.14.c	619	libdebug-0.4.4.c	7,645
rovclock-0.6e.c	1,177	cmigemo-1.2+gh0.20140306.c	7,729
rexima-1.4.c	1,843	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
unhtml-2.3.9.c	2,057	stripcc-0.2.0.c	8,914
elfrc-0.7.c	2,142	xfpt-0.07.c	9,089
jbofihe-0.38.c	2,182	photopc-3.05.c	9,266
delta-2006.08.03.c	2,273	auto-apt-0.3.23ubuntu0.14.04.1.c	11,110
whichman-2.4.c	2,493	glhack-1.2.c	11,237
acpi-1.7.c	2,597	sac-1.9b5.c	11,999
zmakebas-1.2.c	2,606	dict-gcide-0.48.1.c	12,318
forkstat-0.01.04.c	2,710	gzip-spec2000.c	12,980
haskell98-tutorial-200006-2.c	3,161	cutils-1.6.c	14,122
kcc-2.3.c	3,429	mpage-2.5.6.c	14,827
ipip-1.1.9.c	3,605	gnuspool-1.7ubuntu1.c	16,665
acpi-1.4.c	3,814	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
pgpgpg-0.13.c	3,908	libchewing-0.3.5.c	19,262
rsrce-0.2.2.c	3,956	jwhois-3.0.1.c	19,375
unsort-1.1.2.c	4,290	archimedes.c	19,552
hexdiff-0.0.53.c	4,334	gnuplot-4.6.4.c	20,306
acorn-fdisk-3.0.6.c	4,450	phalanx-22+d051004.c	24,099
dvbtune-0.5.ds.c	5,068	gnuchess-5.05.c	28,853
bmf-0.9.4.c	5,451	combine-0.3.3.c	29,508
libbind-6.0.c	5,497	tmndec-3.2.0.c	31,890
mixal-1.08.c	5,570	gbsplay-0.0.91.c	34,002
cmdpack-1.03.c	5,575	parser.c	36,178
picocom-1.7.c	5,613	enscript-1.6.5.c	38,787
xdms-1.3.2.c	5,614	libart-lgpl-2.3.21.c	38,815
dtaus-0.9.c	6,018	flex-2.5.39.c	39,977
device-tree-compiler-1.4.0+dfsg.c	6,033	fwlogwatch-1.2.c	46,601
buildtorrent-0.8.c	6,170	chrony-1.29.c	49,119
e2ps-4.34.c	6,222	wget-1.9.c	54,219
apng2gif-1.5.c	6,522	sn-0.3.8.c	56,227
bwm-ng-0.6.c	6,833	bison-2.4.c	59,955
diffstat-1.58.c	7,077	skyeye-1.2.5.c	85,905
lacheck-1.26.c	7,385	rnv-1.7.10.c	93,858
Total	**160,330**	**Total**	**1,061,661**