Adaptive Static Analysis via Learning with Bayesian Optimization

KIHONG HEO, Seoul National University
HAKJOO OH, Korea University
HONGSEOK YANG, University of Oxford
KWANGKEUN YI, Seoul National University

Building a cost-effective static analyzer for real-world programs is still regarded an art. One key contributor to this grim reputation is the difficulty in balancing the cost and the precision of an analyzer. An ideal analyzer should be adaptive to a given analysis task and avoid using techniques that unnecessarily improve precision and increase analysis cost. However, achieving this ideal is highly nontrivial, and it requires a large amount of engineering efforts.

In this article, we present a new learning-based approach for adaptive static analysis. In our approach, the analysis includes a sophisticated parameterized strategy that decides, for each part of a given program, whether to apply a precision-improving technique to that part or not. We present a method for learning a good parameter for such a strategy from an existing codebase via Bayesian optimization. The learnt strategy is then used for new, unseen programs. Using our approach, we developed partially flow- and context-sensitive variants of a realistic C static analyzer. The experimental results demonstrate that using Bayesian optimization is crucial for learning from an existing codebase. Also, they show that among all program queries that require flow- or context-sensitivity, our partially flow- and context-sensitive analysis answers 75% of them, while increasing the analysis cost only by 3.3× of the baseline flow- and context-insensitive analysis, rather than 40× or more of the fully sensitive version.

CCS Concepts: • **Theory of computation → Program analysis**; • **Computing methodologies → Machine learning approaches**;

Additional Key Words and Phrases: Static program analysis, data-driven program analysis, Bayesian optimization

ACM Reference format:
Kihong Heo, Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2018. Adaptive Static Analysis via Learning with Bayesian Optimization. *ACM Trans. Program. Lang. Syst.* 40, 4, Article 14 (November 2018), 37 pages.
https://doi.org/10.1145/3121135

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09. This work was also supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government(MSIT) (Grant No.2017-0-00184, Self-Learning Cyber Immune Technology Development). This work was partly supported by Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (Grant No. B0717-16-0098).
Authors' addresses: H. Oh (corresponding author), Room 616c, Science Library Bldg, College of Informatics, Korea University, Anam-dong 5-ga, Seongbuk-gu, Seoul 136-713, Korea; email: hakjoo_oh@korea.ac.kr.

0164-0925/2018/11-ART14 $15.00
https://doi.org/10.1145/3121135

1 INTRODUCTION

Although the area of static program analysis has progressed substantially in the past two decades, building a cost-effective static analyzer for real-world programs is still regarded an art. One key contributor to this grim reputation is the difficulty in balancing the cost and the precision of an analyzer. An ideal analyzer should be able to adapt to a given analysis task automatically, and avoid using techniques that unnecessarily improve precision and increase analysis cost. However, designing a good adaptation strategy is highly nontrivial, and it requires a large amount of engineering efforts.

In this article, we present a new approach for building an adaptive static analyzer, which can learn its adaptation strategy from an existing codebase. In our approach, the analyzer includes a *parameterized* strategy that decides, for each part of a given program, whether to apply a precision-improving technique to that part or not. This strategy is defined in terms of a function that scores parts of a program. The strategy evaluates parts of a given program using this function, chooses the top k parts for some fixed k, and applies the precision-improving technique to these parts only. The parameter of the strategy controls this entire selection process by being a central component of the scoring function.

Of course, the success of such an analyzer depends on finding a good parameter for its adaptation strategy. We describe a method for learning such a parameter from an existing codebase using Bayesian optimization; the learnt parameter is then used for new, unseen programs. As typical in other machine-learning techniques, this learning part is formulated as an optimization problem: find a parameter that maximizes the number of queries in the codebase that are proved by the analyzer. This is a challenging optimization problem, because evaluating its objective function involves running the analyzer over several programs and so it is expensive. We present an (approximate) solver for the problem that uses the powerful Bayesian optimization technique and avoids expensive calls to the program analyzer as much as possible.

Note that this learning component involves generalization, but the kind not often seen explicitly in other work on program analysis. It collects information about a program analyzer from given programs, and generalizes this information with the purpose of predicting which parameter would lead to a good adaptation strategy for unseen programs. This *cross-program* generalization of a property of an *analysis* itself is absent in most existing program analyzers. For instance, popular proposals for building adaptive program analyzers, such as counter-example-driven abstraction refinement (e.g., Reference [39]) and parametric program analysis [15, 40], are based on the idea of gathering information about a given program and a verification task by a relatively quick (static or dynamic) program analyzer, and adapting the main analyzer based on this information. The scope of the generalization here is a single program, and the target is properties of this program. It is also mentioning that many prior attempts of using machine-learning techniques in program analysis often fall into this single-program generalization, where generalization is done by existing machine-learning algorithms. Our work suggests the value of studying the new type of cross-program generalization on properties of an analyzer, and provides one concrete instance of such generalization based on Bayesian optimization.

Using our approach, we developed partially flow-sensitive and context-sensitive variants of a realistic C program analyzer. The experimental results confirm that using an efficient optimization solver such as ours based on Bayesian optimization is crucial for learning a good parameter from an existing codebase; a naive approach for learning simply does not scale. When our partially flow- and context-sensitive analyzer was run with a learnt parameter, it answered the 75% of the program queries that require flow- or context-sensitivity, while increasing the analysis cost only by 3.3× of the flow- and context-insensitive analysis, rather than 40× or more of the fully sensitive

version. We also apply our approach for choosing an effective set of widening thresholds, which also shows the great benefit of the approach.

Contributions. We summarize our contributions below:

- We propose a new approach for building a program analysis that can adapt to a given verification task. The key feature of our approach is that it can learn an adaptation strategy from an existing codebase automatically, which can then be applied to new unseen programs.
- We present an effective method for learning an adaptation strategy. Our method uses powerful Bayesian optimization techniques, and reduces the number of expensive program-analysis runs on given programs during the learning process. The performance gain by Bayesian optimization is critical for making our approach practical; without it, learning with medium-to-large programs takes too much time.
- We describe three instance analyses of our approach, which are adaptive variants of our program analyzer for C programs. The first adapts the degree of flow sensitivity of the analyzer, the second adapts context sensitivity of the analyzer, and the last adaptively chooses widening thresholds. In all cases, the experiments show the clear benefits of our approach.

The present article extends the previous version [25] in four ways:

(1) We present a new idea for improving the efficiency of the learning algorithm (Sections 5.4 and 7.4). The technique is based on the idea of ordinal optimization, and reduces the cost of evaluating the objective function of our learning algorithm.
(2) We provide a new instance of our approach and its experimental results (Sections 6.3 and 7.3). We have applied our method to adaptively choosing threshold values for widening of an interval analysis, showing that our method is generally applicable to various analysis instances.
(3) We also show the generality of our approach by applying it to a new client analysis (Section 7.1). In the previous version [25], we evaluated the performance only for a buffer-overrun client. This article shows that our method is also applicable to the detection of null-dereference errors.
(4) Last, we experimentally compare our algorithm with Bayesian optimization with other discrete optimization algorithms such as Basin-hopping [38] and Differential evolution [37] (Section 7.5). The results show that using Bayesian optimization is much more effective than other methods for the program-analysis application, mainly because the objective function in this case is very expensive to evaluate.

2 OVERVIEW

We illustrate our approach using a static analysis with the interval domain. Consider the following program.

```
x=0; y=0; z=1;
x=z;
z=z+1;
y=x;
assert(y>0);
```

The program has three variables (x, y, and z) and the goal of the analysis is to prove that the assertion at line 5 holds.

2.1 Partially Flow-Sensitive Analysis

Our illustration uses a partially flow-sensitive analysis. Given a set of variables V, it tracks the values of selected variables in V flow-sensitively, but for the other variables, it computes global flow-insensitive invariants of their values. For instance, when $V = \{x, y\}$, the analysis computes the following results:

Flow-sensitive		Flow-insensitive
Line	Abstract state	Abstract state
1	$\{x \mapsto [0,0], \quad y \mapsto [0,0]\}$	
2	$\{x \mapsto [1,+\infty], y \mapsto [0,0]\}$	
3	$\{x \mapsto [1,+\infty], y \mapsto [0,0]\}$	$\{z \mapsto [1,+\infty]\}$
4	$\{x \mapsto [1,+\infty], y \mapsto [1,+\infty]\}$	
5	$\{x \mapsto [1,+\infty], y \mapsto [1,+\infty]\}$	

The results are divided into two parts: flow-sensitive and flow-insensitive results. In the flow-sensitive part, the analysis maintains an abstract state at each program point, where each state involves only the variables in V. The information for the other variables (z) is kept in the flow-insensitive state, which is a single abstract state valid for the entire program. Note that this partially flow-sensitive analysis is precise enough to prove the given assertion; at line 5, the analysis concludes that y is greater than 0.

In our example, our $\{x, y\}$ and the entire set $\{x, y, z\}$ are the only choices of V that lead to the proof of the assertion: with any other choice ($V \in \{\emptyset, \{x\}, \{y\}, \{z\}, \{x, z\}, \{y, z\}\}$), the analysis fails to prove the assertion. Our analysis adapts to the program here automatically and picks V. We will next explain how this adaption happens.

2.2 Adaptation Strategy Parameterized with $\mathbf{w}$

Our analysis employs a parameterized strategy (or decision rule) for selecting a set V of variables that will be treated flow-sensitively. The strategy is a function of the form:

$$\mathcal{S}_{\mathbf{w}} : Pgm \to \wp(\mathsf{Var}),$$

which is parameterized by a vector $\mathbf{w}$ of real numbers.

Given a program to analyze, our strategy works in three steps:

(1) We represent all the variables of the program as feature vectors.
(2) We then compute the score of each variable x, which is just the linear combination of the parameter $\mathbf{w}$ and the feature vector of x.
(3) We choose the top-k variables based on their scores, where k is specified by users. In this example, we use $k = 2$.

Step 1: Extracting Features. Our analysis uses a pre-selected set π of features, which are just predicates on variables and summarize syntactic or semantic properties of variables in a given program. For instance, a feature $\pi_i \in \pi$ indicates whether a variable is a local variable of a function or not. These feature predicates are chosen for the analysis, and reused for all programs. The details of the features that we used are given in later sections of this article. In the example of this section, let us assume that our feature set π consists of five predicates:

$$\pi = \{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5\}.$$

Given a program and a feature set π, we can represent each variable x in the program as a feature vector $\pi(x)$:

$$\pi(x) = \langle \pi_1(x), \pi_2(x), \pi_3(x), \pi_4(x), \pi_5(x) \rangle.$$

Suppose that the feature vectors of variables in the example program are as follows:

$$\begin{array}{rcl} \pi(x) & = & \langle 1, 0, 1, 0, 0 \rangle, \\ \pi(y) & = & \langle 1, 0, 1, 0, 1 \rangle, \\ \pi(z) & = & \langle 0, 0, 1, 1, 0 \rangle. \end{array}$$

Step 2: Scoring. Next, we compute the scores of variables based on the feature vectors and the parameter $\mathbf{w}$. The parameter $\mathbf{w}$ is a real-valued vector that has the same dimension as the feature vector, i.e., in this example, $\mathbf{w} \in \mathbb{R}^5$ for $\mathbb{R} = [-1, 1]$. Intuitively, $\mathbf{w}$ encodes the relative importance of each feature.

Given a parameter $\mathbf{w} \in \mathbb{R}^5$, e.g.,

$$\mathbf{w} = \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle, \tag{1}$$

the score of variable x is computed as follows:

$$score(x) = \pi(x) \cdot \mathbf{w}.$$

In our example, the scores of x, y, and z are

$$\begin{array}{l} score(x) = \langle 1, 0, 1, 0, 0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.3, \\ score(y) = \langle 1, 0, 1, 0, 1 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.6, \\ score(z) = \langle 0, 0, 1, 1, 0 \rangle \cdot \langle 0.9, 0.5, -0.6, 0.7, 0.3 \rangle = 0.1. \end{array}$$

Step 3: Choosing Top-k Variables. Finally, we choose the top-k variables based on their scores. For instance, when $k = 2$, we choose variables x and y in our example. As we have already pointed out, this is the right choice in our example, because analyzing the example program with $V = \{x, y\}$ proves the assertion.

2.3 Learning the Parameter w

Manually finding a good parameter $\mathbf{w}$ is difficult. We expect that a program analysis based on our approach uses more than 30 features, so its parameter $\mathbf{w}$ lives in $\mathbb{R}^n$ for some $n \geq 30$. This is a huge search space. It is unrealistic to ask a human to acquire intuition on this space and come up with a right $\mathbf{w}$ that leads to a suitable adaptation strategy for most programs in practice.[1]

The learning part of our approach aims at automatically finding a good $\mathbf{w}$. It takes a codebase consisting of typical programs and searches for a parameter $\mathbf{w}$ that instantiates an adaptation strategy appropriately for programs in the codebase: with this instantiation, a program analysis can prove a large number of queries in these programs.

We explain how our learning algorithm works by using a small codebase that consists of just the following two programs:

[1] In our experiments, all manually chosen parameters lead to strategies that perform much worse than the one automatically learnt by the method in this subsection.

```
a = 0; b = input();
for (a = 0; a < 10; a++);
assert (a > 0);
```
P_1

```
c = d = input();
if (d <= 0) return;
assert (d > 0);
```
P_2

Given this codebase, our learning algorithm looks for $\mathbf{w}$ that makes the analysis prove the two assert statements in P_1 and P_2. Our intention is to use the learnt $\mathbf{w}$ later when analyzing new unseen programs (such as the example in the beginning of this section). We assume that program variables in P_1 and P_2 are summarized by feature vectors as follows:

$$\pi(a) = \langle 0, 1, 1, 0, 1\rangle, \quad \pi(b) = \langle 1, 0, 0, 1, 0\rangle,$$
$$\pi(c) = \langle 0, 1, 0, 0, 1\rangle, \quad \pi(d) = \langle 1, 1, 0, 1, 0\rangle.$$

Simple Algorithm Based on Random Sampling. Let us start with a simple learning algorithm that uses random sampling. Going through this simple algorithm will help a reader to understand our learning algorithm based on Bayesian optimization. The algorithm based on random sampling works in four steps. First, it generates n random samples in the space $\mathbb{R}^5$. Second, for each sampled parameter $\mathbf{w}_i \in \mathbb{R}^5$, the algorithm instantiates the strategy with $\mathbf{w}_i$, runs the static analysis with the variables chosen by the strategy, and records how many assertions in the given codebase are proved. Finally, it chooses the parameter $\mathbf{w}_i$ with the highest number of proved assertions.

The following table shows the results of running this algorithm on our codebase $\{P_1, P_2\}$ with $n = 5$. For each sampled parameter $\mathbf{w}_i$, the table shows the variables selected by the instantiated strategy with $\mathbf{w}_i$ (here, we assume that we choose $k = 1$ variable from each program), and the number of assertions proved in the codebase.

Try	Sample $\mathbf{w}_i$	Decision		#Proved	
		P_1	P_2	P_1	P_2
1	−0.0, 0.7, −0.9, 1.0, −0.7	b	d	0	1
2	0.2, −0.0, −0.8, −0.5, −0.2	b	c	0	0
3	0.4, −0.6, −0.6, 0.6, −0.7	b	d	0	1
4	−0.5, 0.5, −0.5, −0.6, −0.9	a	c	1	0
5	−0.6, −0.8, −0.1, −0.9, −0.2	a	c	1	0

Four parameters achieve the best result, which is to prove one assert statement (either from P_1 or P_2). Among these four, the algorithm returns one of them, such as:

$$\mathbf{w} = \langle -0.0, -0.7, 0.9, 1.0, -0.7\rangle.$$

Note that this is not an ideal outcome; we would like to prove both assert statements. To achieve this ideal, our analysis needs to select variables a from P_1 and d from P_2 and treat them flow-sensitively. But random searching has low probability for finding $\mathbf{w}$ that leads to this variable selection. This shortcoming of random sampling is in a sense expected, and it does appear in practice. As we show in Figure 2, most of the randomly sampled parameters in our experiments perform poorly. Thus, to find a good parameter via random sampling, we need a large number of trials, but each trial is expensive, because it involves running a static analysis over all the programs in a given codebase.

Bayesian Optimization. To describe our algorithm, we need to be more precise about the setting and the objective of algorithms for learning $\mathbf{w}$. These learning algorithms treat a program analysis

and a given codebase simply as a specification of an (objective) function,

$$F : \mathbb{R}^n \to \mathbb{N}.$$

The input to the function is a parameter $\mathbf{w} \in \mathbb{R}^n$, and the output is the number of queries in the codebase that are proved by the analysis. The objective of the learning algorithms is to find $\mathbf{w}^* \in \mathbb{R}^n$ that maximises the function F:

$$\text{Find } \mathbf{w}^* \in \mathbb{R}^n \text{ that maximises } F(\mathbf{w}). \tag{2}$$

Bayesian optimization [3, 18] is a generic algorithm for solving an optimization where an objective function does not have a nice mathematical structure such as gradient and convexity, and evaluating this function is expensive. It aims at minimizing the evaluation of the objective function as much as possible. Notice that our objective function F in (2) lacks good mathematical structures and is expensive to evaluate, so it is a good target of Bayesian optimization. Also, the aim of Bayesian optimization is directly related to the inefficiency of the random-sampling algorithm mentioned above.

The basic structure of Bayesian optimization is similar to the random sampling algorithm. It repeatedly evaluates the objective function with different inputs until it reaches a time limit, and returns the best input found. However, Bayesian optimization diverges from random sampling in one crucial aspect: it builds a probability model about the objective function, and uses the model for deciding where to evaluate the function next. Bayesian optimization builds the model based on the results of the evaluation so far, and updates the model constantly according to the standard rules of Bayesian statistics when it evaluates the objective function with new inputs.

In the rest of this overview, we focus on explaining informally how typical Bayesian optimization builds and uses a probabilistic model, instead of specifics of our algorithm. This will help a reader to see the benefits of Bayesian optimization in our problem. The full description of our learning algorithm is given in Section 5.3.

Assume that we are given an objective function G of type $\mathbb{R} \to \mathbb{R}$. Bayesian optimization constructs a probabilistic model for this unknown function G, where the model expresses the optimizer's current belief about G. The model defines a distribution on functions of type $\mathbb{R} \to \mathbb{R}$ (using so called Gaussian process [26]). Initially, it has high uncertainty about what G is, and assumes that positive outputs or negative outputs are equally possible for G, so the mean (i.e., average) of this distribution is the constant zero function $\lambda x.\, 0$. This model is shown in Figure 1(a), where the large blue region covers typical functions sampled from this model.

Suppose that Bayesian optimization chooses $x = 1.0$, evaluates $G(1.0)$, and get 0.1 as the output. Then, it incorporates this input-output pair, $(1.0, 0.1)$, for G into the model. The updated model is shown in Figure 1(b). It now says that $G(1.0)$ is definitely 0.1, and that evaluating G near 1.0 is likely to give an output similar to 0.1. But it remains uncertain about G at inputs further from 1.0.

Bayesian optimization uses the updated model to decide a next input to use for evaluation. This decision is based on balancing two factors: one for exploiting the model and finding the maximum of G (called exploitation), and the other for evaluating G with an input very different from old ones and refining the model based on the result of this evaluation (called exploration). This balancing act is designed to minimize the number of evaluation of the objective function. For instance, Bayesian optimization now may pick $x = 5.0$ as the next input to try, because the model is highly uncertain about G at this input. If the evaluation $G(5.0)$ gives 0.8, then Bayesian optimization updates the model to one in in Figure 1. Next, Bayesian optimization may decide to use the input $x = 3.0$, because the model predicts that G's output at 3.0 reasonably high on average but it has high uncertainty around this input. If $G(3.0) = 0.65$, then Bayesian optimization updates the model as shown in Figure 1(d). At this point, Bayesian optimization may decide that exploiting

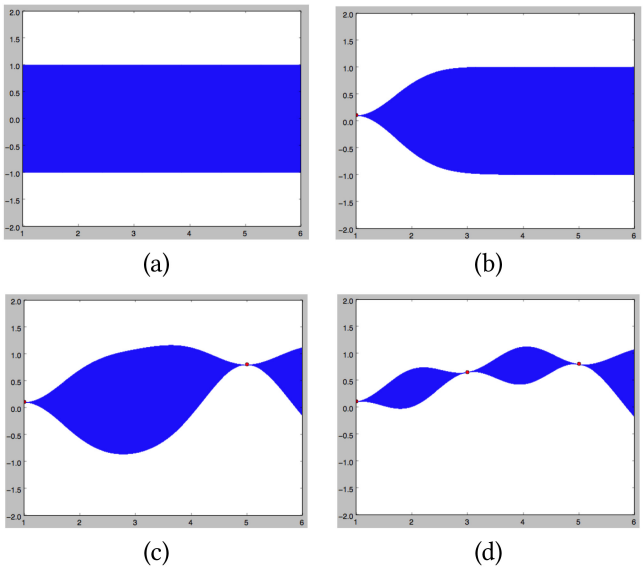


Fig. 1. A typical scenario on how the probabilistic model gets updated during Bayesian optimization.

the model so far outweighs the benefit of exploring G with new inputs, and pick $x = 4.0$, where G is expected to give a high value according to the model. By incorporating all the information about G into the model and balancing exploration and exploitation, Bayesian optimization fully exploits all the available knowledge about G, and minimizes the expensive evaluation of the function G.

3 ADAPTIVE STATIC ANALYSIS

We use a well-known setup for building an adaptive (or parametric) program analysis [15]. In this approach, an analysis has switches for parts of a given program that determine whether these parts should be analyzed with high precision or not. It adapts to the program by turning on these switches selectively according to a fixed strategy.[2] For instance, a partially context-sensitive analysis has switches for call sites of a given program, and use them to select call sites that will be treated with context sensitivity.

Let $P \in \mathbb{P}$ be a program that we would like to analyze. We assume a set $\mathbb{J}_P$ of indices that represent parts of P. For instance, $\mathbb{J}_P$ is the set of program variables in our partially flow-sensitive analysis. We define a set of abstractions as follows:

$$\mathbf{a} \in \mathcal{A}_P = \{0, 1\}^{\mathbb{J}_P}.$$

Abstractions are binary vectors with indices in $\mathbb{J}_P$, and are ordered pointwise:

$$\mathbf{a} \sqsubseteq \mathbf{a}' \iff \forall j \in \mathbb{J}_P.\ \mathbf{a}_j \leq \mathbf{a}'_j.$$

Intuitively, $\mathbb{J}_P$ consists of the parts of P where we have switches for controlling the precision of an analysis. For instance, in a partially context-sensitive analysis, $\mathbb{J}_P$ is the set of procedures or call sites in the program. In our partially flow-sensitive analysis, it denotes the set of program variables that are analyzed flow-sensitively. An abstraction $\mathbf{a}$ is just a particular setting of the

[2]This type of an analysis is usually called parametric program analysis [15]. We do not use this phrase in the article to avoid confusion; if we did, we would have two types of parameters, ones for selecting parts of a given program, and the others for deciding a particular adaption strategy of the analysis.

switches associated with $\mathbb{J}_P$ and determines a program abstraction to be used by the analyzer. Thus, $\mathbf{a}_j = 1$ means that the component $j \in \mathbb{J}_P$ is analyzed, e.g., with context sensitivity or flow sensitivity. We sometimes regard an abstraction $\mathbf{a} \in \mathcal{A}_P$ as a function from $\mathbb{J}_P$ to $\{0, 1\}$, or the following collection of P's parts:

$$\mathbf{a} = \{j \in \mathbb{J}_P \mid \mathbf{a}_j = 1\}.$$

In the latter case, we write $|\mathbf{a}|$ for the size of the collection. The last notation is two constants in $\mathcal{A}_P$:

$$\mathbf{0} = \lambda j \in \mathbb{J}_P.\, 0, \quad \text{and} \quad \mathbf{1} = \lambda j \in \mathbb{J}_P.\, 1,$$

which represent the most imprecise and precise abstractions, respectively. In the rest of this article, we omit the subscript P when there is no confusion.

We assume that a set of assertions is given together with P. The goal of the analysis is to prove as many assertions as possible. An adaptive static analysis is modeled as a function:

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

Given an abstraction $\mathbf{a} \in \mathcal{A}$, $F(P, \mathbf{a})$ returns the number of assertions proved under the abstraction $\mathbf{a}$. Usually, the used abstraction correlates the precision and the performance of the analysis. In this article, we generally assume that using a more refined abstraction is likely to improve the precision of the analysis but increase its cost,[3] which generally holds in static analysis for the C programming languages. However, in other analyses, e.g., for Java, more precise analysis could lead to better scalability as more spurious paths are eliminated during the analysis. Thus, most existing adaptation strategies aim at finding a small $\mathbf{a}$ that makes the analysis prove as many queries as the abstraction $\mathbf{1}$.

3.1 Goal

Our goal is to learn a good adaptation strategy automatically from an existing codebase $\mathbf{P} = \{P_1, \ldots, P_m\}$ (that is, a collection of programs). A learnt strategy is a function of the following type,[4]

$$\mathcal{S} : Pgm \to \mathcal{A},$$

and is used to analyse new, unseen programs P,

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

If the learnt strategy is good, then running the analysis with $\mathcal{S}(P)$ would give results close to those of the most precise abstraction ($F(P, \mathbf{1})$), while incurring the cost at the level of or only slightly above the least precise and hence cheapest abstraction ($F(P, \mathbf{0})$).

In the rest of the article, we explain how we achieve this goal. We first define a parameterized adaptation strategy that scores program parts based on a parameterized linear function and selects high scorers for receiving precise analysis (Section 4). Next, we present a learning algorithm via Bayesian optimization for finding good parameter values from the codebase (Section 5).

[3] If $\mathbf{a} \sqsubseteq \mathbf{a}'$, then we typically have $F(P, \mathbf{a}) \leq F(P, \mathbf{a}')$, but performing $F(P, \mathbf{a}')$ costs more than performing $F(P, \mathbf{a})$.

[4] Strictly speaking, the set of abstractions varies depending on a given program, so a strategy is a dependently typed function and maps a program to one of the abstractions associated with the program. We elide this distinction to simplify presentation.

4 PARAMETERIZED ADAPTATION STRATEGY

In this section, we explain a *parameterized* adaptation strategy, which defines our hypothesis space $\mathcal{H}$ mentioned in the previous section. Intuitively, this parameterized adaptation strategy is a template for all the candidate strategies that our analysis can use when analyzing a given program, and its instantiations with different parameter values form $\mathcal{H}$.

Recall that for a given program P, an adaption strategy chooses a set of components of P that will be analyzed with high precision. As explained in Section 2.2, our parameterized strategy makes this choice in three steps. We formalize these steps next.

4.1 Feature Extraction

Given a program P, our parameterized strategy first represents P's components by so called feature vectors. A *feature* π^k is a predicate on program components:

$$\pi^k_P : \mathbb{J}_P \to \{0, 1\} \text{ for each program } P.$$

For instance, when components in $\mathbb{J}_P$ are program variables, checking whether a variable j is a local variable or not is a feature. Our parameterized strategy requires that a static analysis comes with a collection of features:

$$\pi_P = \left\{\pi^1_P, \ldots, \pi^n_P\right\}.$$

Using these features, the strategy represents each program component j in $\mathbb{J}_P$ as a Boolean vector as follows:

$$\pi_P(j) = \left\langle \pi^1_P(j), \ldots, \pi^n_P(j) \right\rangle.$$

We emphasize that the same set of features is reused for all programs, as long as the same static analysis is applied to them.

As in any other machine-learning approaches, choosing a good set of features is critical for the effectiveness of our learning-based approach. We discuss our choice of features for two instance program analyses in Section 6. According to our experience, finding these features required efforts, but was not difficult, because the used features were mostly well-known syntactic properties of program components.

4.2 Scoring

Next, our strategy computes the scores of program components using a linear function of feature vectors: for a program P,

$$score^{\mathbf{w}}_P : \mathbb{J}_P \to \mathbb{R},$$
$$score^{\mathbf{w}}_P(j) = \pi_P(j) \cdot \mathbf{w}.$$

Here, we assume $\mathbb{R} = [-1, 1]$ and $\mathbf{w} \in \mathbb{R}^n$ is a real-valued vector with the same dimension as the feature vector. The vector $\mathbf{w}$ is the parameter of our strategy, and determines the relative importance of each feature when our strategy chooses a set of program components.

We extend the score function to abstractions $\mathbf{a}$:

$$score^{\mathbf{w}}_P(\mathbf{a}) = \sum_{j \in \mathbb{J}_P \wedge \mathbf{a}(j)=1} score^{\mathbf{w}}_P(j),$$

which sums the scores of the components chosen by $\mathbf{a}$.

Table 1. The Minimal Flow-sensitivity for Interval Abstract Domain Is Significantly Small

Program	#Var	Flow-insensitivity		Flow-sensitivity		Minimal flow-sensitivity	
		proved	time(s)	proved	time(s)	time(s)	size
time-1.7	353	36	0.1	37	0.4	0.1	1 (0.3%)
spell-1.0	475	63	0.1	64	0.8	0.1	1 (0.2%)
barcode-0.96	1,729	322	1.1	335	5.7	1.0	5 (0.3%)
archimedes	2,467	423	5.0	1110	28.1	4.2	104 (4.2%)
tar-1.13	5,244	301	7.4	469	316.1	8.9	75 (1.4%)
TOTAL	10,268	1,145	13.7	2,015	351.1	14.3	186 (1.8%)

#Var shows the number of program variables (abstract locations) in the programs. Proved and time show the number of proved buffer-overrun queries in the programs and the running time of each analysis. Minimal flow-sensitivity proves exactly the same queries as the flow-sensitivity while taking analysis time comparable to that of flow-insensitivity.

4.3 Selecting Top-k Components

Finally, our strategy selects program components based on their scores, and picks an abstraction accordingly. Given a fixed $k \in \mathbb{R}\,(0 \leq k \leq 1)$, it chooses $\lfloor k \times |\mathbb{J}_P| \rfloor$ components with highest scores. For instance, when $k = 0.1$, it chooses the top 10% of program components. Then, the strategy returns an abstraction $\mathbf{a}$ that maps these chosen components to 1 and the rest to 0.

Let $\mathcal{A}^k$ be the set of abstractions that contains $\lfloor k \times |\mathbb{J}_P| \rfloor$ elements when viewed as a set of $j \in \mathbb{J}_P$ with $\mathbf{a}_j = 1$:

$$\mathcal{A}^k = \{\mathbf{a} \in \mathcal{A} \mid |\mathbf{a}| = \lfloor k \times |\mathbb{J}_P| \rfloor\}.$$

Formally, our parameterized strategy $\mathcal{S}_{\mathbf{w}} : Pgm \to \mathcal{A}^k$ is defined as follows:

$$\mathcal{S}_{\mathbf{w}}(P) = \underset{\mathbf{a} \in \mathcal{A}^k_P}{\operatorname{argmax}}\, score^{\mathbf{w}}_P(\mathbf{a}). \tag{3}$$

That is, given a program P and a parameter $\mathbf{w}$, it selects an abstraction $\mathbf{a} \in \mathcal{A}^k$ with maximum score.

A reader might wonder which k value should be used. In our case, we set k close to 0 (e.g., $k = 0.1$) so that our strategy choose a small and cheap abstraction. Typically, this in turn entails a good performance of the analysis with the chosen abstraction.

Using such a small k is based on a conjecture that for many verification problems, the sizes of minimal abstractions sufficient for proving these problems are significantly small. One evidence of this conjecture is given by Reference [15], who presented algorithms to find minimal abstractions (the coarsest abstraction sufficient to prove all the queries provable by the most precise abstraction) and showed that, in a pointer analysis used for discharging queries from a race detector, only a small fraction (0.4–2.3%) of call-sites are needed to prove all the queries provable by 2-CFA analysis. We also observed that the conjecture holds for flow-sensitive numeric analysis and buffer-overrun queries. We implemented Liang et al.'s ActiveCoarsen algorithm, and found that the minimal flow-sensitivity involves only 0.2–4.2% of total program variables, which means that we can achieve the precision of full flow-sensitivity with a cost comparable to that of flow-insensitivity (see Table 1).

5 OUR LEARNING ALGORITHM

We present our approach for learning a parameter of the adaptation strategy. We formulate the learning process as an optimization problem, and solve it efficiently via the techniques of Bayesian optimization and ordinal optimization.

5.1 The Optimization Problem

In our approach, learning a parameter from a codebase $\mathbf{P} = \{P_1, \ldots, P_m\}$ corresponds to solving the following optimization problem. Let n be the number of features of our strategy in Section 4.1. Then, the optimization problem is described as follows:

$$\text{Find } \mathbf{w}^* \in \mathbb{R}^n \text{ that maximizes } \sum_{P_i \in \mathbf{P}} F(P_i, \mathcal{S}_{\mathbf{w}^*}(P_i)). \tag{4}$$

That is, the goal of the learning is to find $\mathbf{w}^*$ that maximizes the number of proved queries on programs in $\mathbf{P}$ when these programs are analyzed with the strategy $\mathcal{S}_{\mathbf{w}^*}$. However, solving this optimization problem exactly is impossible. The objective function involves running static analysis F over the entire codebase, which is very expensive to evaluate in realistic settings. Furthermore, it lacks a good structure—it is not convex and does not even have a derivative. Thus, we lower our aim slightly, and look for an approximate answer, i.e., a parameter $\mathbf{w}$ that makes the objective function close to its maximal value.

5.2 Learning via Random Sampling

We begin with a simple method for approximately solving the problem in (4), based on random sampling (Algorithm 1). This random-sampling method works by repeatedly applying the following steps:

(1) Pick a parameter $\mathbf{w} \in \mathbb{R}^n$ randomly, where n is the number of features.
(2) Instantiate the parameterized strategy $\mathcal{S}$ with the $\mathbf{w}$.
(3) Run the static analysis with the instantiation $\mathcal{S}_{\mathbf{w}}$ over the codebase, and compute the number s of proved queries:

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

We repeat this procedure until we exhaust our budget for time, and choose the $\mathbf{w}$ with the highest s among all samples. Algorithm 1 describes this method.

Although the method is simple and easy to implement, it is extremely inefficient according to our experience. The inefficiency is twofold. First, in our experiments, most of randomly sampled parameters have poor qualities, failing to prove the majority of queries on programs in $\mathbf{P}$ (Section 7.1). Second, evaluating the objective function in a realistic setting is very expensive, because it involves running static analysis over the entire codebase. Thus, to find a good parameter using this method, we need to evaluate the expensive objective function many times, which is not feasible in reality.

In the rest of this section, we improve the performance of this baseline algorithm with two techniques:

- We use Bayesian optimization (Section 5.3) to find a good parameter without evaluating the objective function many times.
- We use ordinal optimization (Section 5.4) to reduce the cost of evaluating the objective function while retaining the final quality of the learning algorithm.

Note that the two techniques are orthogonal to each other; the algorithm intertwined with two techniques is better than the one with a single technique only.

ALGORITHM 1: Learning via Random Sampling

Input: codebase $\mathbf{P}$ and static analysis F
Output: best parameter $\mathbf{w} \in \mathbb{R}^n$ found

$\mathbf{w}_{max} \leftarrow$ sample from $\mathbb{R}^n$ ($\mathbb{R} = [-1, 1]$)
$max \leftarrow \sum_{P_i \in \mathbf{P}} F(P_i, \mathcal{S}_{\mathbf{w}_{max}}(P_i))$
repeat
  $\mathbf{w} \leftarrow$ sample from $\mathbb{R}^n$
  $s = \sum_{P_i \in \mathbf{P}} F(P_i, \mathcal{S}_{\mathbf{w}}(P_i))$
  if $s > max$ **then**
    $max \leftarrow s$
    $\mathbf{w}_{max} \leftarrow \mathbf{w}$
  end if
until timeout
return $\mathbf{w}_{max}$

ALGORITHM 2: Learning via Bayesian optimization

Input: codebase $\mathbf{P}$ and static analysis F
Output: best parameter $\mathbf{w} \in \mathbb{R}^n$ found

$\Theta \leftarrow \emptyset$
for $i \leftarrow 1, t$ **do** ▷ random initialization
  $\mathbf{w} \leftarrow$ sample from $\mathbb{R}^n$
  $s = \sum_{P_i \in \mathbf{P}} F(P_i, \mathcal{S}_{\mathbf{w}}(P_i))$
  $\Theta \leftarrow \Theta \cup \{\langle \mathbf{w}, s \rangle\}$
end for
$\langle \mathbf{w}_{max}, max \rangle \leftarrow \langle \mathbf{w}, s \rangle \in \Theta$ s.t. $\forall \langle \mathbf{w}', s' \rangle \in \Theta.\ s' \leq s$
repeat
  update the model $\mathcal{M}$ by incorporating new data Θ (i.e., compute the posterior distribution of $\mathcal{M}$ given Θ, and set $\mathcal{M}$ to this distribution)
  $\mathbf{w} = \text{argmax}_{\mathbf{w} \in \mathbb{R}^n} acq(\mathbf{w}, \Theta, \mathcal{M})$
  $s = \sum_{P_i \in \mathbf{P}} F(P_i, \mathcal{S}_{\mathbf{w}}(P_i))$
  $\Theta \leftarrow \{\langle \mathbf{w}, s \rangle\}$
  if $s > max$ **then**
    $max \leftarrow s$
    $\mathbf{w}_{max} \leftarrow \mathbf{w}$
  end if
until timeout
return $\mathbf{w}_{max}$

5.3 Learning via Bayesian Optimization

Bayesian optimization is a powerful method for solving difficult optimization problems where objective functions are expensive to evaluate [3] and do not have good structures, such as derivative. Typically, optimizers for such a problem work by evaluating its optimization function with many different inputs and returning the input with the best output. The key idea of Bayesian optimization is to reduce this number of evaluations by constructing and using a probabilistic model for the objective function. The model defines a probability distribution on functions, predicts what the objective function looks like (i.e., mean of the distribution), and describes uncertainty on its

prediction (i.e., variance of the distribution). The model gets updated constantly during the optimization process (according to Bayes's rule), such that it incorporates the results of all the previous evaluations of the objective function. The purpose of the model is, of course, to help the optimizer pick a good input to evaluate next, good in the sense that the output of the evaluation is large and reduces uncertainty of the model considerably. We sum up our short introduction to Bayesian optimization by repeating its two main components in our program-analysis application:

(1) Probabilistic model $\mathcal{M}$: Initially, this model $\mathcal{M}$ is set to capture a prior belief on properties of the objective function in Equation (4), such as its smoothness. During the optimization process, it gets updated to incorporate the information about all previous evaluations.[5]
(2) Acquisition function *acq*: Given $\mathcal{M}$, this function gives each parameter $\mathbf{w}$ a score that reflects how good the parameter is. This is an easy-to-optimize function that serves as a proxy for our objective function in Equation (4) when our optimizer chooses a next parameter to try. The function encodes a success measure on parameters $\mathbf{w}$ that balances two aims: evaluating our objective function with $\mathbf{w}$ should gives a large value (often called exploitation), and at the same time help us to refine our model $\mathcal{M}$ substantially (often called exploration).

Algorithm 2 shows our learning algorithm based on Bayesian optimization. At lines 2–5, we first perform random sampling for t times, and stores the pairs of parameter $\mathbf{w}$ and score s in Θ (line 5). At line 7, we pick the best parameter and score in Θ. The main loop is at lines from 8 to 17. At line 9, we build the probabilistic model $\mathcal{M}$ from the collected data Θ. At line 10, we select a parameter $\mathbf{w}$ by maximizing the acquisition function.[6] This takes some computation time but is insignificant compared to the cost of evaluating the expensive objective function (running static analysis over the entire codebase). Next, we run the static analysis with the selected parameter $\mathbf{w}$, and update the data (line 12). The loop repeats until we run out of our fixed time budget, at which point the algorithm returns the best parameter found.

Algorithm 2 leaves open the choice of a probabilistic model and an acquisition function, and its performance depends on making a right choice. We have found that a popular standard option for the model and the acquisition function works well for us—the algorithm with this choice outperforms the naive random sampling method substantially. Concretely, we used the Gaussian Process (GP) [26] for our probabilistic model, and the expected improvement (EI) [3] for the acquisition function.[7]

A Gaussian Process (GP) is a well-known probabilistic model for functions to real numbers. Due to its flexibility and tractability, GP is currently the most popular model for estimating real-valued functions [3]. In our setting, these functions are maps from parameters to reals, with the type $\mathbb{R}^n \to \mathbb{R}$. Also, a GP G is a function-valued random variable such that for all $o \in \mathbb{N}$ and parameters $\mathbf{w}_1, \ldots, \mathbf{w}_o \in \mathbb{R}^n$, the results of evaluating F at these parameters,

$$\langle G(\mathbf{w}_1), \ldots, G(\mathbf{w}_o) \rangle,$$

[5] In the jargon of Bayesian optimization or Bayesian statistics, the initial model is called a prior distribution, and its updated versions are called posterior distributions.
[6] We used the limited-memory BFGS (L-BFGS) algorithm for computing the argmax of the acquisition function. This is an approximation algorithm that follows the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm.
[7] We also tried other options for the acquisition function such as probability of improvement (PI) and upper confidence bound (UCB) [3]. Compared to the method with EI, the other methods yield only less than 1% degradation in terms of proven queries.

are distributed according to the o-dimensional Gaussian distribution[8] with mean $\mu \in \mathbb{R}^o$ and covariance matrix $\Sigma \in \mathbb{R}^{o\times o}$, both of which are determined by two hyperparameters to the GP. The first hyperparameter is a mean function $m : \mathbb{R}^n \to \mathbb{R}$, and it determines the mean μ of the output of G at each input parameter:

$$\mu(\mathbf{w}) = m(\mathbf{w}) \text{ for all } \mathbf{w} \in \mathbb{R}^n.$$

The second hyperparameter is a symmetric function $k : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, called kernel, and it specifies the smoothness of G: for each pair of parameters $\mathbf{w}$ and $\mathbf{w}'$, $k(\mathbf{w}, \mathbf{w}')$ describes how close the outputs $G(\mathbf{w})$ and $G(\mathbf{w}')$ are. If $k(\mathbf{w}, \mathbf{w}')$ is positive and large, then $G(\mathbf{w})$ and $G(\mathbf{w}')$ have similar values for most random choices of G. However, if $k(\mathbf{w}, \mathbf{w}')$ is near zero, the values of $G(\mathbf{w})$ and $G(\mathbf{w}')$ do not exhibit such a close relationship. In our experiments, we adopted the common choice for hyperparameters and initialized a GP as follows:

$$m = \lambda \mathbf{w}.\, 0, \qquad k(\mathbf{w}, \mathbf{w}') = \exp(-\|\mathbf{w} - \mathbf{w}'\|^2/2).$$

Incorporating data to the GP G with m and k above is done by computing the so called posterior of G with respect to the data. Suppose that we have evaluated our objective function with parameters $\mathbf{w}_1, \ldots, \mathbf{w}_t$ and obtained the values of the function $s_1, \ldots, s_t$. The value s_i represents the number of proved queries when the static analysis is run with parameter $\mathbf{w}_i$ over the given codebase. Let $\Theta = \{\langle \mathbf{w}_i, s_i \rangle \mid 1 \le i \le n\}$. The posterior of G with respect to Θ is a probability distribution obtained by updating the one for G using information in Θ. It is well-known that this posterior distribution $p(G \mid \Theta)$ is again a GP and has the following mean and kernel functions:

$$\begin{aligned} m'(\mathbf{w}) &= \mathbf{k}\mathbf{K}^{-1}\mathbf{s}^T, \\ k'(\mathbf{w}, \mathbf{w}') &= k(\mathbf{w}, \mathbf{w}') - \mathbf{k}\mathbf{K}^{-1}\mathbf{k}'^T, \end{aligned}$$

where

$$\begin{aligned} \mathbf{k} &= [k(\mathbf{w}, \mathbf{w}_1)\ k(\mathbf{w}, \mathbf{w}_2)\ \ldots\ k(\mathbf{w}, \mathbf{w}_t)], \\ \mathbf{k}' &= [k(\mathbf{w}', \mathbf{w}_1)\ k(\mathbf{w}', \mathbf{w}_2)\ \ldots\ k(\mathbf{w}', \mathbf{w}_t)], \\ \mathbf{s} &= [s_1\ s_2\ \ldots\ s_t], \\ \mathbf{K} &= \begin{bmatrix} k(\mathbf{w}_1, \mathbf{w}_1) & \ldots & k(\mathbf{w}_1, \mathbf{w}_t) \\ \vdots & \ddots & \vdots \\ k(\mathbf{w}_t, \mathbf{w}_1) & \ldots & k(\mathbf{w}_t, \mathbf{w}_t) \end{bmatrix}. \end{aligned}$$

Figure 1 shows the outcomes of three posterior updates pictorially. It shows four regions that contain most of functions sampled from GPs.

The acquisition function for expected improvement (EI) is defined as follows:

$$acq(\mathbf{w}, \Theta, \mathcal{M}) = \mathbb{E}[\max(F(\mathbf{w}) - s_{\max}, 0) \mid \Theta]. \tag{5}$$

The expected value here is defined w.r.t. the probabilisitic model, GP, and the previously seen data Θ. Here $s_{\max}$ is the maximum score seen in the data Θ so far (i.e., $s_{\max} = \max\{s_i \mid \exists \mathbf{w}_i.\, \langle \mathbf{w}_i, s_i \rangle \in \Theta\}$), and G is a random variable distributed according to the GP posterior with respect to Θ and is our model for the objective function. The formula $\max(G(\mathbf{w}) - s_{\max}, 0)$ in Equation (5) measures the improvement in the maximum score when the objective function is evaluated at $\mathbf{w}$. The right hand side of the equation computes the expectation of this improvement, justifying the name

[8] A random variable $\mathbf{x}$ with value in $\mathbb{R}^o$ is a o-dimensional Gaussian random variable with mean $\mu \in \mathbb{R}^o$ and covariance matrix $\Sigma \in \mathbb{R}^{o\times o}$ if it has the following probability density:

$$p(\mathbf{x}) = (2\pi)^{-\frac{o}{2}} \times |\Sigma|^{-\frac{1}{2}} \times \exp\left(-\frac{(\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu)}{2}\right).$$

"expected improvement." The further discussion on this acquisition function can be found in Section 2.3 of Reference [3].

5.4 Learning with Ordinal Optimization

Now, we apply the idea of ordinal optimization [6, 12] to reduce the cost of evaluating the objective function. Although the learning algorithm via Bayesian optimization is much more efficient than random sampling, it still requires costly evaluation of the objective function. For the evaluation, it runs static analysis over the entire codebase, which is often infeasible in realistic settings. However, note that our aim is not to obtain the accurate value of each evaluation, but to choose one that is relatively better than other possible alternatives. In this case, ordinal optimization helps to reduce the cost of the objective function with little compromise on the final quality of the parameters found.

A key idea of ordinal optimization is that order is much easier to estimate than cardinal values. Let us illustrate it in the context of our learning algorithm. Without loss of generality, the essence of the learning algorithm can be described as follows:

(1) Sample two parameters $\mathbf{w}_1$ and $\mathbf{w}_2$.
(2) Compute their scores $s_1 = J(\mathbf{w}_1)$ and $s_2 = J(\mathbf{w}_2)$, where J is the objective function:

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

(3) Choose $\mathbf{w}_1$ if $s_1 \geq s_2$ and $\mathbf{w}_2$ otherwise.

Note that what is important in this entire process is the order of the performance values (s_1 and s_2), not the concrete values of them. Therefore, we focus on estimating the order between parameters without exerting to compute the accurate performance values. To this end, we use an approximate version $\hat{F}$ of the original static analyzer F such that the order of the estimated values from $\hat{F}$ is almost same as the order of the actual values from F; that is,

$$\hat{F}(\mathbf{w}_1) \leq \hat{F}(\mathbf{w}_2) \text{ strongly implies that } F(\mathbf{w}_1) \leq F(\mathbf{w}_2), \tag{6}$$

and vice versa. We substitute $\hat{F}$ for F in the learning algorithm, which eases the computational burden of the original objective function. Thanks to the order-preserving approximation in Equation (6), the quality of the parameters found by $\hat{F}$ is as good as that of the parameters found by F.

We obtain $\hat{F}$ from F by randomly choosing $\lfloor |\mathbf{a}| * r \rfloor$ components (rather than all of them in $\mathbf{a}$, i.e., $|\mathbf{a}|$), where $r \in [0, 1]$ is a fixed real number that encodes the degree of the approximation: r is the ratio of the number of actual elements that are analyzed with high precision to the number of the total elements in the given abstraction. Then, we run the static analysis by giving high precision only to these components in $\mathbb{J}_P$.

6 INSTANCE ANALYSES

In this section, we present three instance analyses of our approach that adapt the degrees of flow-sensitivity, context-sensitivity, and widening thresholds, respectively.

6.1 Adaptive Flow-Sensitive Analysis

We define a class of partially flow-sensitive analyses and describe the features used in adaptation strategies for these analyses.

A Class of Partially Flow-Sensitive Analyses. Given a program P, let $(\mathbb{C}, \rightarrow)$ be its control flow graph, where $\mathbb{C}$ is the set of nodes (program points) and $(\rightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ denotes the control flow relation of the program.

An analysis that we consider uses an abstract domain that maps program points to abstract states:

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

Here an abstract state $s \in \mathbb{S}$ is a map from abstract locations (namely, program variables, structure fields, and allocation sites) to values:

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

For each program point c, the analysis comes with a function $f_c : \mathbb{S} \to \mathbb{S}$ that defines the abstract semantics of the command at c.

We assume that the analysis is formulated based on an extension of the sparse-analysis framework [23]. Before going into this formulation, let us recall the original framework for sparse analyses. Let $\mathsf{D}(c) \subseteq \mathbb{L}$ and $\mathsf{U}(c) \subseteq \mathbb{L}$ be the def and use sets at program point $c \in \mathbb{C}$. Using these sets, define a relation $(\rightsquigarrow) \subseteq \mathbb{C} \times \mathbb{L} \times \mathbb{C}$ for data dependency:

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

A way to read $c_0 \overset{l}{\rightsquigarrow} c_n$ is that c_n depends on c_0 on location l. This relationship holds when there exists a path $[c_0, c_1, \ldots, c_n]$ such that l is defined at c_0 and used at c_n, but it is not re-defined at any of the intermediate points c_i. A sparse analysis is characterized by the following abstract transfer function $F : \mathbb{D} \to \mathbb{D}$:

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

This analysis is fully flow-sensitive in that it constructs data dependencies for every abstract location and tracks all these dependencies accurately.

We extend this sparse-analysis framework such that an analysis is allowed to track data dependencies only for abstract locations in some set $L \subseteq \mathbb{L}$, and to be flow-sensitive only for these locations. For the remaining locations (i.e., $\mathbb{L} \setminus L$), we use results from a quick flow-insensitive pre-analysis [23], which we assume given. The results of this pre-analysis form a state $s_I \in \mathbb{S}$, and are stable (i.e., pre-fixpoint) at all program points:

$$\forall c \in \mathbb{C}. \; f_c(s_I) \sqsubseteq s_I.$$

The starting point of our extension is to define the data-dependency with respect to L:

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

The main modification lies in a new requirement that in order for $c_0 \overset{l}{\rightsquigarrow}_L c_n$ to hold, the location l should be included in the set L. With this notion of data dependency, we next define an abstract transfer 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 says that when we collect an abstract state right before c, we use the flow-insensitive result $s_I(l)$ for a location not in L, and follow the original treatment for those in L. An analysis in our extension computes $\mathsf{lfp}_{X_0} F_L$, where the initial $X_0 \in \mathbb{D}$ is built by associating the

results of the flow-insensitive analysis (i.e., values of s_I) with all locations not selected by L (i.e., $\mathbb{L} \setminus L$):

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

Note that L determines the degree of flow-sensitivity. For instance, when $L = \mathbb{L}$, the analysis becomes an ordinary flow-sensitive sparse analysis. However, when $L = \emptyset$, the analysis is just a flow-insensitive analysis. The set L is what we call abstraction in Section 3: abstraction locations in $\mathbb{L}$ form $\mathbb{J}_P$ in that section, and subsets of these locations, such as L, are abstractions there, which are expressed in terms of sets, rather than Boolean functions. Our approach provides a parameterized strategy for selecting the set L that makes the analysis comparable to the flow-sensitive version for precision and to the flow-insensitive one for performance. In particular, it gives a method for learning parameters in that strategy.

Features. The features for our partially flow-sensitive analyses describe syntactic or semantic properties of abstract locations, namely, program variables, structure fields, and allocation sites. Note that this is what our approach instructs, because these locations form the set $\mathbb{J}_P$ in Section 3 and are parts of P where we control the precision of an analysis.

In our implementation, we used 45 features shown in Table 2, which describe how program variables, structure fields or allocation sites are used in typical C programs. When picking these features, we decided to focus on expressiveness, and included a large number of features, instead of trying to choose only important features. Our idea was to let our learning algorithm automatically find out such important ones among our features.

Our features are grouped into Type A and Type B in Table 2. A feature of Type A describes a simple, atomic property for a program variable, a structure field or an allocation site, e.g., whether it is a local variable or not. A feature of Type B, however, describes a slightly complex usage pattern, and is expressed as a combination of atomic features. Type B features have been designed by manually observing typical usage patterns of variables in the benchmark programs. For instance, feature 34 was developed after we observed the following usage pattern of variables:

```
int x; // local variable
if (x < 10)
  ... = malloc (x);
```

It says that x is a local variable, and gets compared with a constant and passed as an argument to a function that does memory allocation. Note that we included these Type B features not because they are important for flow-sensitivity. We included them to increase expressiveness, because our linear learning model with Type A features only cannot express such usage patterns. Deciding whether they are important for flow-sensitivity or not is the job of the learning algorithm.

Our Feature Engineering Process. In this work, we manually designed the features as follows. To identify a feature, we begin with a program that contains a query requiring flow-sensitivity to prove. Given a program P, we can easily collect such queries by running flow-sensitive and flow-insensitive analyses on P and compare the proved queries by these analyses. Suppose we identify the following program and a query from this procedure:

```
1 int* a = malloc(10);
2 int i = 0;
3 while (1) {
4   y = unknown();
5   z = unknown();
```

Table 2. Features for Partially Flow-sensitive Analysis

Type	#	Features
A	1	local variable
	2	global variable
	3	structure field
	4	location created by dynamic memory allocation
	5	defined at one program point
	6	location potentially generated in library code
	7	assigned a constant expression (e.g., x = c1 + c2)
	8	compared with a constant expression (e.g., x < c)
	9	compared with an other variable (e.g., x < y)
	10	negated in a conditional expression (e.g., if (!x))
	11	directly used in malloc (e.g., malloc(x))
	12	indirectly used in malloc (e.g., y = x; malloc(y))
	13	directly used in realloc (e.g., realloc(x))
	14	indirectly used in realloc (e.g., y = x; realloc(y))
	15	directly returned from malloc (e.g., x = malloc(e))
	16	indirectly returned from malloc
	17	directly returned from realloc (e.g., x = realloc(e))
	18	indirectly returned from realloc
	19	incremented by one (e.g., x = x + 1)
	20	incremented by a constant expr. (e.g., x = x + (1+2))
	21	incremented by a variable (e.g., x = x + y)
	22	decremented by one (e.g., x = x - 1)
	23	decremented by a constant expr (e.g., x = x - (1+2))
	24	decremented by a variable (e.g., x = x - y)
	25	multiplied by a constant (e.g., x = x * 2)
	26	multiplied by a variable (e.g., x = x * y)
	27	incremented pointer (e.g., p++)
	28	used as an array index (e.g., a[x])
	29	used in an array expr. (e.g., x[e])
	30	returned from an unknown library function
	31	modified inside a recursive function
	32	modified inside a local loop
	33	read inside a local loop
B	34	$1 \wedge 8 \wedge (11 \vee 12)$
	35	$2 \wedge 8 \wedge (11 \vee 12)$
	36	$1 \wedge (11 \vee 12) \wedge (19 \vee 20)$
	37	$2 \wedge (11 \vee 12) \wedge (19 \vee 20)$
	38	$1 \wedge (11 \vee 12) \wedge (15 \vee 16)$
	39	$2 \wedge (11 \vee 12) \wedge (15 \vee 16)$
	40	$(11 \vee 12) \wedge 29$
	41	$(15 \vee 16) \wedge 29$

(Continued)

Table 2. Continued

Type	#	Features
	42	$1 \wedge (19 \vee 20) \wedge 33$
	43	$2 \wedge (19 \vee 20) \wedge 33$
	44	$1 \wedge (19 \vee 20) \wedge \neg 33$
	45	$2 \wedge (19 \vee 20) \wedge \neg 33$

Features of Type A denote simple syntactic or semantic properties for abstract locations (that is, program variables, structure fields, and allocation sites). Features of Type B are various combinations of simple features, and express patterns that variables are used in programs.

```
6   if (y < z)
7     ...;
8   if (i < 10)
9     a[i] = 0; // buffer-overrun query
10   i = i + 1;
11 }
```

Note that a flow-sensitive analysis is able to prove the safety of buffer-access at line 9 while a flow-insensitive analysis fails to do so. Once we identify such a target program and a query, we remove all irrelevant statements from the program. For instance, variables y and z are irrelevant to the query (no dependencies on the query), so we remove all lines with y and z and obtain the following reduced program:

```
1 a = malloc(10);
2 i = 0;
3 while (1) {
4   if (i < 10)
5     a[i] = 0; // buffer-overrun query
6     i = i + 1;
7 }
```

This code snippet now contains only the statements on which the query depends. From this code, we extract syntactic and semantic properties of involved program variables. For instance, we know that the variable i, which is used as the index of the array-access, is a local variable in this program, generating the feature #1 in our list (Table 2). Also, we observe that i is compared with a constant at line 4 and incremented by one, from which we generate the features #8 and #19, respectively. Similarly, we can create the features #32 (modified inside a local loop), #33 (read inside a local loop), and #28 (used as an array index). We also generate the features for the variable a from the code: #15 (directly returned from malloc) and #29 (used as an array expression).

This way, we can create the *positive features* of program variables, which describe properties of variables that help the analysis to prove queries. In a similar manner, we can also generate *negative features*, which describe the properties of variables involved in unprovable queries. Our process is manual, but the same principle (i.e., identifying positive/negative queries, generating minimal programs, and extract features from them) can be applied to design the features for other program analyses (e.g., Tables 3 and 4).

6.2 Adaptive Context-Sensitive Analysis

Another example of our approach is adaptive context-sensitive analyses. Assume we are given a program P. Let *Procs* be the set of procedures in P. The adaptation strategy of such an analysis

Table 3. Features for Partially Context-sensitive Analysis

Type	#	Features
A	1	leaf function
	2	function containing malloc
	3	function containing realloc
	4	function containing a loop
	5	function containing an if statement
	6	function containing a switch statement
	7	function using a string-related library function
	8	write to a global variable
	9	read a global variable
	10	write to a structure field
	11	read from a structure field
	12	directly return a constant expression
	13	indirectly return a constant expression
	14	directly return an allocated memory
	15	indirectly return an allocated memory
	16	directly return a reallocated memory
	17	indirectly return a reallocated memory
	18	return expression involves field access
	19	return value depends on a structure field
	20	return void
	21	directly invoked with a constant
	22	constant is passed to an argument
	23	invoked with an unknown value
	24	functions having no arguments
	25	functions having one argument
	26	functions having more than one argument
	27	functions having an integer argument
	28	functions having a pointer argument
	29	functions having a structure as an argument
B	30	$2 \wedge (21 \vee 22) \wedge (14 \vee 15)$
	31	$2 \wedge (21 \vee 22) \wedge \neg(14 \vee 15)$
	32	$2 \wedge 23 \wedge (14 \vee 15)$
	33	$2 \wedge 23 \wedge \neg(14 \vee 15)$
	34	$2 \wedge (21 \vee 22) \wedge (16 \vee 17)$
	35	$2 \wedge (21 \vee 22) \wedge \neg(16 \vee 17)$
	36	$2 \wedge 23 \wedge (16 \vee 17)$
	37	$2 \wedge 23 \wedge \neg(16 \vee 17)$
	38	$(21 \vee 22) \wedge \neg 23$

selects a subset Pr of procedures of P, and instructs the analysis to treat only the ones in Pr context-sensitively: calling contexts of each procedure in Pr are treated separately by the analysis. This style of implementing partial context-sensitivity is intuitive and well-studied, so we omit the details and just mention that our implementation used one such analysis in [22] after minor modification. Note that these partially context-sensitive analyses are instances of the adaptive static

Table 4. Features for Widening-with-thresholds

Type	#	Features
A	1	used in array declarations (e.g., a[c])
	2	used in memory allocation (e.g., malloc(c))
	3	used in the righthand-side of an assignment (e.g., x = c)
	4	used with the less-than operator (e.g., x < c)
	5	used with the greater-than operator (e.g., x > c)
	6	used with ≤ (e.g., x ≤ c)
	7	used with ≥ (e.g., x ≥ c)
	8	used with the equality operator (e.g., x == c)
	9	used with the not-equality operator (e.g., x ! = c)
	10	used within other conditional expressions (e.g., x < c+y)
	11	used inside loops
	12	used in return statements (e.g., return c)
	13	constant zero
B	14	$(1 \vee 2) \wedge 3$
	15	$(1 \vee 2) \wedge (4 \vee 5 \vee 6 \vee 7)$
	16	$(1 \vee 2) \wedge (8 \vee 9)$
	17	$(1 \vee 2) \wedge 11$
	18	$(1 \vee 2) \wedge 12$
	19	$13 \wedge 3$
	20	$13 \wedge (4 \vee 5 \vee 6 \vee 7)$
	21	$13 \wedge (8 \vee 9)$
	22	$13 \wedge 11$
	23	$13 \wedge 12$

analysis in Section 3; the set *Procs* corresponds to $\mathbb{J}_P$, and *Pr* is what we call an abstraction in that section.

For partial context-sensitivity, we used 38 features in Table 3. Since our partially context-sensitive analysis adapts by selecting a subset of procedures, our features are predicates over procedures, i.e., $\pi^k : \mathit{Procs} \to \mathbb{B}$. As in the flow-sensitivity case, we used both atomic features (Type A) and compound features (Type B), both describing properties of procedures, e.g., whether a given procedure is a leaf in the call graph.

The previous two analyses can be combined to an adaptive analysis that controls both flow-sensitivity and context-sensitivity. The combined analysis adjusts the level of abstraction at abstract locations and procedures. This means that its $\mathbb{J}_J$ set consists of abstract locations and procedures, and its abstractions are just subsets of these locations and procedures. The features of the combined analysis are obtained similarly by putting together the features for our previous analyses. This combined abstractions and features enable our learning algorithm to find a more complex adaptation strategy that considers both flow-sensitivity and context-sensitivity at the same time. This strategy helps the analysis to use its increased flexibility efficiently. In Section 7.2, we report our experience with experimenting the combined analysis.

6.3 Adaptive Widening-with-Thresholds

The last instance of our approach is an analysis that adaptively chooses widening thresholds. We describe the technique of widening-with-thresholds, and present the features used in its adaptation

strategy. For clarity, we describe the technique in the context of the interval domain [5], but the general idea behind the technique is applicable regardless of the underlying abstract domain.

Widening in Static Analysis. We begin with a short introduction to widening in static analysis. In abstract interpretation, a static analysis is specified by a pair of abstract domain $\mathbb{D}$ and abstract semantic function $F : \mathbb{D} \to \mathbb{D}$. The goal of the analysis is to compute an upper bound A of the least fixed point of F, i.e.,

$$A \sqsupseteq \bigsqcup_{i \geq 0} F^i(\bot).$$

When the height of the domain $\mathbb{D}$ is finite, the least fixed point computation eventually stabilizes. However, when $\mathbb{D}$ has an infinite height, the algorithm may not terminate, and a widening operator should be used to complete the analysis in finite steps. A widening operator $\bigtriangledown : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ is a binary operator that has the two properties:

(1) It is an upper bound operator, i.e., $\forall a, b \in \mathbb{D}.\ (a \sqsubseteq a \bigtriangledown b)\ \wedge\ (b \sqsubseteq a \bigtriangledown b)$.
(2) For all increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ in $\mathbb{D}$, the chain $y_0 = x_0, y_{i+1} = y_i \bigtriangledown x_{i+1}$ is eventually stabilizes after finite steps.

With a widening operator $\bigtriangledown$, the upper bound A is computed by $A = \lim_{i \geq 0} X_i$, where chain X_i is defined as follows:

$$\begin{array}{rll} X_0 &= \bot, & \\ X_{i+1} &= X_i & F(X_i) \sqsubseteq X_i \\ &= X_i \bigtriangledown F(X_i) & \text{otherwise.} \end{array}$$

The abstract interpretation framework guarantees that the above chain is always finite and its limit (i.e., $\lim_{i \geq 0} X_i$) is an upper bound of the least fixed point of F [5]. For instance, a simple widening operator for the interval domain works as follows: (For brevity, we do not consider the bottom interval.)

$$[a, b] \bigtriangledown [c, d] = [(c < a ? -\infty : a), (b < d ? +\infty : b)]$$

That is, the widening operator extrapolates any unstable bounds simply to infinity. For instance, $[1, 4] \bigtriangledown [1, 7] = [1, +\infty]$.

Widening with Thresholds. The idea of widening-with-thresholds is to bound the extrapolation of the widening using a pre-defined set of thresholds. For instance, suppose we are given a set $T = \{8, 9\}$ of thresholds. Then, applying widening $[1, 4] \bigtriangledown^T [1, 7]$ with thresholds $T = \{8, 9\}$ gives interval $[1, 8]$, instead of $[1, +\infty]$. Here, threshold 8 is used, because it is the smallest value in T, which is greater than 7. If the result is still unstable in the subsequent iteration, then the next smallest value in T, i.e., 9, is used to bound the widening.

Formally, the widening-with-thresholds technique for the interval domain is defined as follows. We assume that a set $T \subseteq \mathbb{Z} \cup \{-\infty, +\infty\}$ of thresholds is given. Without loss of generality, let us assume that $T = \{t_1, t_2, \ldots, t_n\}$, $t_1 < t_2 < \cdots < t_n$, $t_1 = -\infty$, and $t_n = +\infty$. The widening operator parameterized by T is defined as follows:

$$[a, b] \bigtriangledown^T [c, d] = ([a, b] \bigtriangledown [c, d]) \sqcap \bigsqcap \{[t_l, t_u] \mid t_l, t_u \in T\ \wedge\ t_l \leq \min(a, c)\ \wedge\ t_u \geq \max(b, d)\}$$

For instance, when $T = \{-\infty, 0, 5, +\infty\}$,

$$[2, 3] \bigtriangledown^T [1, 4] = [-\infty, +\infty] \sqcap \bigsqcap \{[0, 5], [-\infty, +\infty]\} = [-\infty, +\infty] \sqcap [0, 5] = [0, 5].$$

Note that the precision and scalability of the technique is entirely controlled by the choice of T. For instance, when $T = \emptyset$, the analysis degenerates into the analysis with the standard widening, which is very hasty in extrapolating unstable bounds to infinities. However, when $L = U$ (assume that U is the universal set for thresholds such as the set of all integer constants in the given

program), the analysis becomes the most precise one, where the analysis attempts to bound the widening with every integer constant in the program. The set T is what we call abstraction in Section 3; the set of integers in U forms $\mathbb{J}_P$ in that section, and subsets of these integers are abstractions there. Our goal is to choose a good set $T \subseteq U$ of thresholds, which makes the analysis comparable to the analysis with $T = U$ for precision and to the analysis with $T = \emptyset$ for scalability. In particular, we aim to learn such a T from an existing codebase.

Features. The features for the widening-with-thresholds technique describe properties of integer constants in the program; a feature is a predicate over integers, i.e., $\pi^k : \mathbb{Z} \to \mathbb{B}$. In our implementation, we used 23 syntactic features shown in Table 4, which describe how integer constants are used in typical C programs, e.g., whether the constant is used as an argument of memory allocation functions.

7 EXPERIMENTS

Following our recipe in Section 6, we instantiated our approach for partial flow-sensitivity, partial context-sensitivity, and partial widening-with-thresholds. We implemented these instantiations in Sparrow, an interval domain–based static analyzer for real-world C programs [24]. In this section, we report the results of our experiments with these implementations.

- Section 7.1 applies our learning method to flow-sensitive analysis and evaluates its performance with two clients, buffer-overflow and null-dereference detection.
- Section 7.2 evaluates our approach for context-sensitivity.
- Section 7.3 evaluates our approach for widening-with-thresholds, i.e., for choosing threshold values for widening of the interval analysis.
- Section 7.4 shows the effectiveness of the ordinal optimization idea proposed in Section 5.4.
- Section 7.5 compares our algorithm with Bayesian optimization to other discrete optimization algorithms.

7.1 Partial Flow-Sensitivity

Setting. We implemented a partial flow-sensitive analysis in Section 6.1 by modifying Sparrow, which supports the full C language and has been being developed for the past seven years [24]. This baseline analyzer tracks both numeric and pointer-related information simultaneously in its fixpoint computation. For numeric values, it uses the interval abstract domain, and for pointer values, it uses an allocation-site-based heap abstraction. The analysis is field-sensitive (i.e., separates different structure fields) and flow-sensitive, but it is not context-sensitive. We applied the sparse analysis technique [23] to improve the scalability.

By modifying the baseline analyzer, we implemented a partially flow-sensitive analysis, which controls its flow-sensitivity according to a given set of abstract locations (program variables, structure fields and allocation sites) as described in Section 6.1. We also implemented our learning algorithm based on Bayesian optimization. Our implementations were tested against 30 open source programs from GNU and Linux packages (Table 7).

The key questions that we would like to answer in our experiments are whether our learning algorithm produces a good adaptation strategy and how much it gets benefited from Bayesian optimization. To answer the first question, we followed a standard method in the machine-learning literature, called cross validation. We randomly divide the 30 benchmark programs into 20 training programs and 10 test programs. An adaptation strategy is learned from the 20 training programs, and tested against the remaining 10 test programs. We repeated this experiment for five times. The results of each trial are shown in Table 5. In these experiments, we set $k = 0.1$, which means that flow-sensitivity is applied to only the 10% of total abstract locations (i.e., program variables,

Table 5. Effectiveness of Our Method for Flow-sensitivity

	Training			
	FI	FS	partial FS	
Trial	prove	prove	prove	quality
1	6,383	7,316	7,089	75.7%
2	5,788	7,422	7,219	87.6%
3	6,148	7,842	7,595	85.4%
4	6,138	7,895	7,599	83.2%
5	7,343	9,150	8,868	84.4%
TOTAL	31,800	39,625	38,370	**84.0%**

	Testing								
	FI		FS			partial FS			
	prove	s	prove	s	cost	prove	s	quality	cost
1	2,788	48	4,009	627	13.2×	3,692	78	74.0%	1.6×
2	3,383	55	3,903	531	9.6×	3,721	93	65.0%	1.7×
3	3,023	49	3,483	1,898	38.6×	3,303	99	60.9%	2.0×
4	3,033	38	3,430	237	6.2×	3,286	51	63.7%	1.3×
5	1,828	28	2,175	577	20.5×	2,103	54	79.3%	1.9×
TOTAL	14,055	218	17,000	3,868	**17.8×**	16,105	374	**69.6%**	**1.7×**

Prove: The number of proved queries in each analysis (FI: Flow-insensitivity, FS: Flow-sensitivity, Partial FS: Partial Flow-sensitivity). Quality: The ratio of proved queries among the queries that require flow-sensitivity. Cost: Cost increase compared to the FI analysis (buffer-overrun client).

structure fields and allocation sites). We compared the performance of a flow-insensitive analysis (FI), a fully flow-sensitive analysis (FS) and our partially flow-sensitive variant (partial FS). To answer the second question, we compared the performance of the Bayesian optimization-based learning algorithm against the random sampling method.

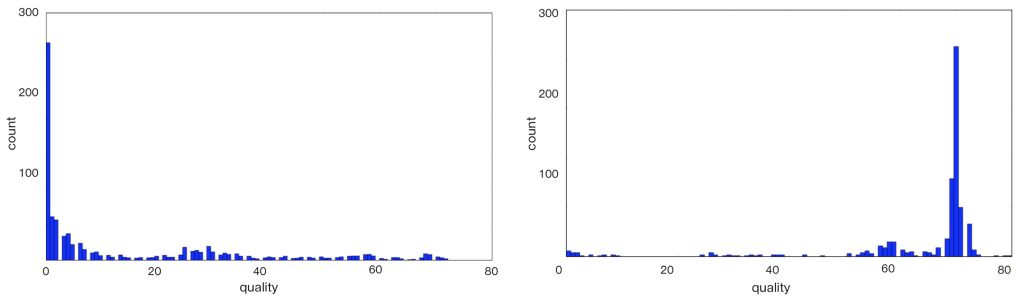
Learning. Table 5 shows the results of the training and test phases for all the five trials. In total, the flow-insensitive analysis (FI) proved 31,800 buffer-overrun queries in the 20 training programs, while the fully flow-sensitive analysis (FS) proved 39,625 queries. During the learning phase, our algorithm found a parameter $\mathbf{w}$. On the training programs, the analysis with $\mathbf{w}$ proved, on average, 84.0% of FS-only queries, that is, queries that were handled only by the flow-sensitive analysis (FS). Finding such a good parameter for training programs, let alone unseen test ones, is highly nontrivial. As shown in Table 2, the number of parameters to tune at the same time is 45 for flow-sensitivity. Manually searching for a good parameter $\mathbf{w}$ for these 45 parameter over 18 training programs is simply impossible. In fact, we tried to do this manual search in the early stage of this work, but most of our manual trials failed to find any useful parameter (Figure 2).

Figure 2 compares our learning algorithm based on Bayesian optimisation against the one based on random sampling. It shows the two distributions of the qualities of tried parameters $\mathbf{w}$ (recorded in the x axis), where the first distribution uses parameters tried by random sampling over a fixed time budget (12h) and the second, by Bayesian optimisation over the same budget. By the quality of $\mathbf{w}$, we mean the percentage of FS-only queries proved by the analysis with $\mathbf{w}$. The results for random sampling (Figure 2(a)) confirm that the space for adaptation parameters $\mathbf{w}$ for partial flow-sensitivity is nontrivial; most of the parameters do not prove any queries. As a result, random sampling wastes most of its execution time by running the static analysis that does not prove any

Table 6. Effectiveness for Flow-Sensitivity with the Null-Dereference Client

	Training			
	FI	FS	partial FS	
Trial	prove	prove	prove	quality
1	8,956	10,080	9,847	79.3%
2	8,470	9,366	9,289	91.4%
3	8,267	9,394	9,198	82.6%
4	8,174	9,194	9,078	88.6%
5	8,925	10,142	9,999	88. %
TOTAL	42,792	48,176	47,411	**85.8%**

	Testing								
	FI		FS			partial FS			
	prove	s	prove	s	cost	prove	s	quality	cost
1	2,026	33	2,409	270	8.2×	2,390	87	95.0%	2.6×
2	2,512	34	3,123	291	8.5×	2,918	86	66.4%	2.5×
3	2,715	18	3,095	1,233	70.5×	3,052	80	88.7%	4.6×
4	2,808	35	3,295	109	3.1×	3,124	63	64.9%	1.8×
5	2,057	12	2,347	270	22.9×	2,257	53	69.0%	4.5×
TOTAL	12,118	131	14,269	2,172	**16.6×**	13,741	368	**75.5%**	**2.8×**



(a) Random sampling (b) Bayesian optimisation

Fig. 2. Comparison of Bayesian optimisation with random sampling.

FS-only queries. This shortcoming is absent in Figure 2(b) for Bayesian optimisation. In fact, most parameters found by Bayesian optimisation led to adaptation strategies that prove about 70% of FS-only queries. Figure 3 shows how the best qualities found by Bayesian optimisation and random sampling change as the learning proceeds. The results compare the first 30 evaluations for the first training set of our experiments, which show that Bayesian optimisation finds a better parameter (63.5%) with fewer evaluations. The random sampling method converged to the quality of 45.2%.

Testing. For each of the five trials, we tested a parameter learnt from 20 training programs, against 10 programs in the test set. The results of this test phase are given in Table 5, and they show that the analysis with the learnt parameters has a good precision/cost balance. In total, for 10 test programs, the flow-insensitive analysis (FI) proved 14,055 queries, while the full flow-sensitive one (FS) proved 17,000 queries. The partially flow-sensitive version with a learnt adaptation strategy proved on average 69.6% of the FS-only queries. To do so, our partially flow-sensitive analysis

Table 7. Benchmark Programs

Programs	LOC
cd-discid-1.1	421
time-1.7	1,759
unhtml-2.3.9	2,057
spell-1.0	2,284
mp3rename-0.6	2,466
ncompress-4.2.4	2,840
pgdbf-0.5.0	3,135
cam-1.05	5,459
e2ps-4.34	6,222
sbm-0.0.4	6,502
mpegdemux-0.1.3	7,783
barcode-0.96	7,901
bzip2	9,796
bc-1.06	16,528
gzip-1.2.4a	18,364
unrtf-0.19.3	19,015
archimedes	19,552
coan-4.2.2	28,280
gnuchess-5.05	28,853
tar-1.13	30,154
tmndec-3.2.0	31,890
agedu-8642	32,637
gbsplay-0.0.91	34,002
flake-0.11	35,951
enscript-1.6.5	38,787
mp3c-0.29	52,620
tree-puzzle-5.2	62,302
icecast-server-1.3.12	68,564
aalib-1.4p5	73,412
rnv-1.7.10	93,858
TOTAL	743,394

increases the cost of the FI analysis only moderately (by 1.7×), while the FS analysis increases the analysis cost by 17.8×.

However, the results show that the analyses with the learnt parameters are generally less precise in the test set than the training set. For the five trials, our method has proved, on average, 84.0% of FS-queries in the training set and 69.6% in the test set.

Top-10 Features. The learnt parameter identified the features that are important for flow-sensitivity. Because our learning method computes the score of abstract locations based on linear combination of features and parameter $\mathbf{w}$, the learnt parameter $\mathbf{w}$ means the relative importance of features.

Figure 4 shows the 10 most important features identified by our learning algorithm from ten trials (including the five trials in Table 5 as well as additional five ones). For instance, in the first trial, we found that the most important features were #19, 32, 1, 4, 28, 33, 29, 3, 43, 18 in Table 2.

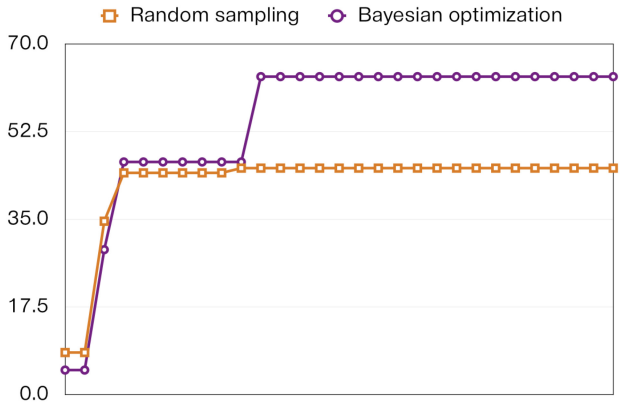


Fig. 3. Comparison of Bayesian optimisation with random sampling.

	Trials									
rank	1	2	3	4	5	6	7	8	9	10
1	# 19	# 19	# 19	# 19	# 19	# 11	# 11	# 11	# 13	# 19
2	# 32	# 32	# 32	# 32	# 32	# 19	# 19	# 19	# 19	# 28
3	# 1	# 28	# 37	# 1	# 1	# 28	# 24	# 28	# 28	# 32
4	# 4	# 33	# 40	# 27	# 4	# 12	# 26	# 12	# 32	# 7
5	# 28	# 29	# 31	# 4	# 28	# 1	# 28	# 1	# 26	# 3
6	# 33	# 18	# 1	# 28	# 7	# 32	# 32	# 4	# 7	# 33
7	# 29	# 8	# 39	# 7	# 15	# 26	# 18	# 42	# 45	# 24
8	# 3	# 14	# 27	# 9	# 33	# 21	# 43	# 23	# 3	# 20
9	# 43	# 37	# 20	# 6	# 29	# 7	# 36	# 32	# 33	# 40
10	# 18	# 9	# 4	# 15	# 3	# 45	# 7	# 6	# 35	# 8

Fig. 4. (Left) Top-10 features (for flow-sensitivity) identified by our learning algorithm for ten trials. Each entry denotes the feature numbers shown in Table 2. (Right) Counts of each feature (x-axis) that appears in the top-10 features during the ten trials. Features #19 and #32 are in top-10 for all trials. The results have been obtained with 20 training programs.

These features say that accurately analysing, for instance, variables incremented by one (#19) or modified inside a local loop (#32), and local variables (#1) are important for cost-effective flow-sensitive analysis. The histogram on the right shows the number of times each feature appears in the top-10 features during the ten trials. In all trials, features #19 (variables incremented by one) and #32 (variables modified inside a local loop) are included in the top-10 features. Features #1 (local variables), #4 (locations created by dynamic memory allocation), #7 (location generated in library code), and #28 (used as an array index) appear more than five times across the ten trials. We also identified top-10 features when trained with a smaller set of programs. Figure 6 shows the results with 10 training programs. In this case, features #1 (local variables), #7 (assigned a constant expression), #9 (compared with another variable), #19 (incremented by one), #28 (used as an array index), and #32 (modified inside a local loop) appeared more than five times across ten trials.

The automatically selected features generally coincided with our intuition on when and where flow-sensitivity helps. For instance, the following code (taken from barcode-0.96) shows a typical situation where flow-sensitivity is required:

```
int mirror[7];
int i = unknown;
```

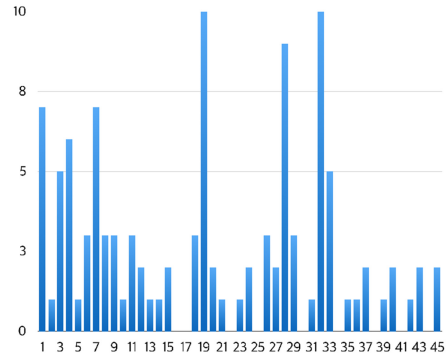


Fig. 5. Top-10 feature frequencies with 20 training programs.

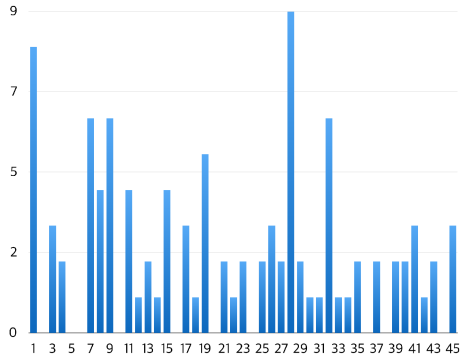


Fig. 6. Top-10 feature frequencies with 10 training programs.

```
for (i = 1; i < 7; i++)
 if (mirror[i-1] == '1') ...
```

Because variable `i` is initially unknown and is incremented in the loop, a flow-insensitive interval analysis cannot prove the safety of buffer access at line 3. However, if we analyze variable `i` flow-sensitively, we can prove that `i-1` at line 3 always has a value less than 7 (the size of `mirror`). Note that, according to the top-10 features, variable `i` has a high score in our method, because it is a local variable (#1), modified inside a local loop (#32), and incremented by one (#19).

The selected features also provided novel insights that contradicted our conjecture. When we manually identified important features in the early stage of this work, we conjectured that feature #10 (variables negated in a conditional expression) would be a good indicator for flow-sensitivity, because we found the following pattern in the program under investigation (spell-1.0):

```
int pos = unknown;
if (!pos)
 path[pos] = 0;
```

Although `pos` is unknown at line 1, its value at line 3 must be 0, because `pos` is negated in the condition at line 2. However, after running our algorithm over the entire codebase, we found that this pattern happens only rarely in practice, and that feature #10 is actually a strong indicator for flow-"insensitivity."

Effectiveness of the type-B Features. We evaluated the effectiveness of our method without the type-B features. In our experience, the type-B features is less important than the type-A features. Without the type-B features, however, the results are highly unstable depending on training and test sets. Our partially flow-sensitive analysis without type-B features yields similar performance on average: 67.7% of the FS-only queries. However, the individual results vary a lot: It proved 84.9% in the trial 1, yet only 46.8% in the trial 5.

Robustness of the Features. We evaluated the robustness of our features under semantic-preserving transformations. We transformed the benchmark programs with (1) common compiler optimizations (partial evaluation, constant folding and dead code elimination) and (2) three-address code using the CIL frontend [20].

According to the experiments, semantic-preserving transformations do not sacrifice the performance much. First, the simple compiler optimizations made only marginal difference. It is because our analysis and analysis are not that sensitive to different syntactic representations of constants and dead codes. Meanwhile, the three-address code degrades the quality to 58.8%. This transformation results in lower-level C codes that are harder to analyze in general. For example, in the three-address code, field accesses are transformed to byte offsets that our analyzer does not handle precisely.

Sensitivity to the Benchmarks. We evaluated the sensitivity of our method to the characteristics of the benchmark programs. We tried different cross-validation by collecting test sets according to their characteristics. Among the 30 benchmark programs, we derived two test sets: text-processing programs and mathematical programs (compression, scientific computation, and game).

The proposed learning method is not that sensitive to training/test sets. The partially flow-sensitive analysis proved 93.8% of the FS-only queries in the text-processing programs and 70.4% in the mathematical programs; both of them are higher than the average quality of the randomly selected test sets.

Comparison to a Simple Random Search. We tried an exhaustive random search on only a few programs and evaluated the found weight vector. We performed random search on two small programs (time-1.7 and unhtml-2.3.9) for 12h. However, the random search could not find a generally good weight. The best wight vector during the random search is not effective to apply to unseen programs. In the experiments, the derived partially flow-sensitive analysis only proved 42.5% of the FS-only queries.

Performance with a Null-Dereference Client. We also evaluated the effectiveness of our method with null-dereference checking. A null-dereference query is of the form `assert(p != NULL)`, checking if pointer `p` could have the null value. Table 6 shows the results of the training and test phases for five trials for null-dereference queries. In these experiments, we used the same features that were used for buffer-overrun queries. In total, the flow-insensitive analysis (FI) proved 42,792 null-dereference queries and the flow-sensitive analysis (FS) proved 48,176 queries during the training phase (with 20 programs), proving 85.8% of FS-only queries. In the test phase, the flow-insensitive analysis proved 12,118 queries, while the full flow-sensitive analysis proved 14,269 queries. Our partially flow-sensitive analysis with the parameter found in the training phase proved 13,741 queries, proving 75.5% of the FS-only queries. These results show that our learning approach effectively generalizes to other client analyses as well.

Comparison with the Impact Pre-analysis Approach. Recently, Oh et al. [22] proposed to run a cheap pre-analysis, called impact pre-analysis, and to use its results for deciding which parts of a given program should be analysed precisely by the main analysis [22]. We compared our approach with Oh et al.'s proposal on partial flow sensitivity. Following Oh et al.'s recipe [22], we

implemented a impact pre-analysis that is fully flow-sensitive but uses a cheap abstract domain, in fact, the same one as in Reference [22], which mainly tracks whether integer variables store non-negative values or not. Then, we built an analysis that uses the results of this pre-analysis for achieving partially flow-sensitivity.

In our experiments with the programs in Table 5, the analysis based on the impact pre-analysis proved 80% of queries that require flow-sensitivity, and spent 5.5× more time than flow-insensitive analysis. Our new analysis of this article, however, proved 70% and spent only 1.7× more time. Furthermore, in our approach, we can easily obtain the analysis that is selective both in flow- and context-sensitivity (Section 7.2), which is is nontrivial in the pre-analysis approach.

7.2 Adding Partial Context-Sensitivity

As another instance of our approach, we implemented an adaptive analysis for supporting both partial flow-sensitivity and partial context-sensitivity. Our implementation is an extension of the partially flow-sensitive analysis, and follows the recipe in Section 6.2. Its learning part finds a good parameter of a strategy that adapts flow-sensitivity and context-sensitivity simultaneously. This involves 83 parameters in total (45 for flow-sensitivity and 38 for context-sensitivity) and is a more difficult problem (as an optimisation problem as well as as a generalisation problem) than the one for partial flow-sensitivity only.

Setting. We implemented context-sensitivity by inlining. All the procedures selected by our adaptation strategy get inlined. To avoid code explosion by such inlining, we inlined only relatively small procedures. Specifically, to be inlined in our experiments, a procedure should have less-than-100 basic blocks. The results of our experiments are shown in Table 8. In the table, FSCS means the fully flow-sensitive and fully context-sensitive analysis, where all procedures with less-than-100 basic blocks are inlined. FICI denotes the fully flow-insensitive and context-insensitive analysis. Our analysis (partial FSCS) represents the analysis that selectively applies both flow-sensitivity and context-sensitivity.

Results. The results show that our learning algorithm finds a good parameter $\mathbf{w}$ of our adaption strategy. The learnt $\mathbf{w}$ generalises well to unseen programs, and leads to an adaptation strategy that achieves high precision with reasonable additional cost. In training programs, FICI proved 26,904 queries, and FSCS proved 39,555 queries. With a learnt parameter $\mathbf{w}$ on training programs, our partial FSCS proved 79.3% of queries that require flow-sensitivity or context-sensitivity or both. More importantly, the parameter $\mathbf{w}$ worked well for test programs, and proved 81.2% of queries of similar kind. Regarding the cost, our partial FSCS analysis increased the cost of the FICI analysis only by 3.0×, while the fully flow- and context-sensitive analysis (FSCS) increased it by 80.5×.

7.3 Adaptive Widening-with-Thresholds

The last instance is an interval analysis that adaptively chooses threshold values for widening (Section 6.3). Among all constant integers in the program, we choose 10% of them with highest scores. Table 9 shows that our learning algorithm finds a very good parameter for choosing widening thresholds both in training and test phases. In the training phase, the interval analysis with no thresholds proved 39,943 (buffer-overrun) queries, whereas the analysis with the entire threshold set (i.e. the set of all constant integers in program texts) proved 40,502 queries. Our partial analysis proved 40,465 queries (93.4%). In the test phase, the analysis without thresholds proved 17,322 queries. With full threshold set, the analysis can prove 18,213 queries, but it increases the analysis cost by 9.5×. Our selective technique proved 96.2% of queries that require thresholds while only increasing the cost by 1.5×.

Table 8. Effectiveness for Flow-sensitivity + Context-sensitivity (buffer-overrun client)

	Training			
	FI	FS	partial FS	
Trial	prove	prove	prove	quality
1	6,383	9,237	8,674	80.3%
2	5,788	8,287	7,598	72.4%
3	6,148	8,737	8,123	76.3%
4	6,138	9,883	8,899	73.7%
5	7,343	10,082	10,040	98.5%
TOTAL	31,800	46,226	43,334	**80.0%**

	Testing								
	FI		FS			partial FS			
	prove	s	prove	s	cost	prove	s	quality	cost
1	2,788	46	4,275	5,425	118.2×	3,907	187	75.3%	4.1×
2	3,383	57	5,225	4,495	79.4×	4,597	194	65.9%	3.4×
3	3,023	48	4,775	5,235	108.8×	4,419	150	79.7%	3.1×
4	3,033	38	3,629	1,609	42.0×	3,482	82	75.3%	2.1×
5	1,828	30	2,670	7,801	258.3×	2,513	104	81.4%	3.4×
TOTAL	14,055	219	20,574	24,565	**112.1×**	18,918	717	**74.6%**	**3.3×**

Table 9. Effectiveness for Widening-with-thresholds with the Buffer-overrun Client

	Training			
	without thresholds	full thresholds	partial thresholds	
Trial	prove	prove	prove	quality
1	7,403	7,455	7,445	80.8%
2	7,444	7,516	7,510	91.7%
3	7,971	8,045	8,042	95.9%
4	7,934	8,026	8,013	85.9%
5	9,191	9,460	9,455	98.1%
TOTAL	39,943	40,502	40,465	**93.4%**

	Testing								
	without thresholds		full thresholds			partial thresholds			
	prove	s	prove	s	cost	prove	s	quality	cost
1	4,050	424	4,288	3,856	9.1×	4,271	613	92.9%	1.4×
2	4,009	351	4,227	2,247	6.4×	4,221	443	97.2%	1.3×
3	3,482	938	3,698	12,062	12.9×	3,695	1493	98.6%	1.6×
4	3,519	201	3,717	1,817	9.0×	3,713	311	98.0%	1.5×
5	2,262	358	2,283	1,679	4.7×	2,279	483	81.0%	1.3×
TOTAL	17,322	2,273	18,213	21,661	**9.5×**	18,179	3343	**96.2%**	**1.5×**

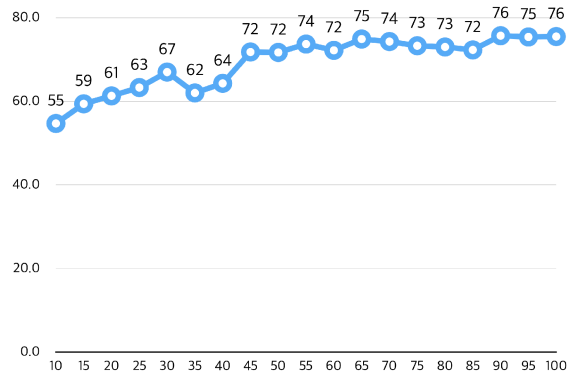


Fig. 7. Quality of the learning algorithm with different r values.

7.4 Performance of Ordinal Optimization

In Section 5.4, we incorporated ordinal optimization into our learning algorithm, where the degree of approximation is controlled by the parameter r. The parameter r is a real number ranging from 0 to 1. If r is 1, then the cost of estimating performance is the same as that of measuring the exact performance value by analyzing all the program components in an abstraction with high precision. If r is 0, then the cost of estimating performance is identical to that of evaluating the exact performance value by not analyzing any component in the abstraction with high precision, often resulting in poor quality. Therefore, we need to choose the proper value for r. We want r to be as small as possible so that we can lower the overall cost of learning, while maintaining the quality of analysis (i.e., precision of analysis) as high as possible.

Figure 7 shows the quality of the learning algorithm with ordinal optimization as r increases. For each r, we ran our learning algorithm on a set of 20 programs and obtained the quality of the best parameter found. We performed these experiments 10 times and averaged the results. Note that the quality drops by only 2 percent (from 76% to 74%) even when r is reduced to half of its original value (from 1 to 0.5). This experiment provides empirical evidence of the reliability of ordinal optimization in learning adaptation strategies of static analysis.

7.5 Comparison to Other Optimization Algorithms

We compared Bayesian optimization with two popular optimization algorithms: Basin-hopping [38] and Differential evolution [37]. Basin-hopping performs a stochastic search akin to simulated annealing and Differential evolution is an evolutionary algorithm.

In our case, these algorithms were extremely ineffective: with the same training time as Bayesian optimization, Basin-hopping proved only 1.4% of the FS-only queries in the learning phase and 2.3% in the testing phase. The Differential evolution could not find parameters better than initial random ones. The main reason is that these algorithms basically require evaluating the objective function many times (e.g., the evolutionary algorithm needs a large set of populations to work), which is infeasible in our case. However, the main strength of Bayesian optimization is that it is well-suited for expensive objective functions. This is why we chosen Bayesian optimization for learning program analysis parameters.

8 RELATED WORK AND DISCUSSION

Parametric Program Analysis. Parametric program analyses refer to a program analysis that is equipped with a class of program abstractions and analyzes a given program by selecting

abstractions from this class appropriately. The original idea of parametric program analysis was proposed in Reference [15], where the goal is to learn a minimal abstraction that is a coarsest abstraction capable of proving all provable queries. Reference [15] showed that such a minimal abstraction is significantly small. Our work exploits this fact and provides a learning approach for finding a good abstraction. Parametric analyses commonly adopt counter-example-guided abstraction refinement, and selects a program abstraction based on the feedback from a failed analysis run [1, 4, 7, 8, 10, 11, 39, 40]. Some exceptions to this common trend are to use the results of dynamic analysis [9, 19] or pre-analysis [22, 35] for finding a good program abstraction.

However, automatically finding such a strategy is not what they are concerned with, while it is the main goal of our work. All of the previous approaches focus on designing a good fixed strategy that chooses a right abstraction for a given program and a given query. A high-level idea of our work is to parameterize these adaptation (or abstraction-selection) strategies, not just program abstractions, and to use an efficient learning algorithm (such as Bayesian optimization) to find right parameters for the strategies. One interesting research direction is to try our idea with existing parametric program analyses.

For instance, our method can be combined with the impact pre-analysis [22] to find a better strategy for selective context-sensitivity. In Reference [22], context-sensitivity is selectively applied by receiving a guidance from a pre-analysis. The pre-analysis is an approximation of the main analysis under full context-sensitivity. Therefore, it estimates the impact of context-sensitivity on the main analysis, identifying context-sensitivity that is likely to benefit the final analysis precision. One feature of this approach is that the impact estimation of the pre-analysis is guaranteed to be realized at the main analysis (Proposition 1 in Reference [22]). However, this impact realization does not guarantee the proof of queries; some context-sensitivity is inevitably applied even when the queries are not provable. Also, because the pre-analysis is approximated, the method may not apply context-sensitivity necessary to prove some queries. Our method can be used to reduce these cases; we can find a better strategy for selective context-sensitivity by using the pre-analysis result as a semantic feature together with other (syntactic/semantic) features for context-sensitivity.

Use of Machine Learning in Program Analysis. Several machine-learning techniques have been used for various problems in program analysis. Researchers noticed that many machine-learning techniques share the same goal as program abstraction techniques, namely, to generalise from concrete cases, and they tried these machine-learning techniques to obtain sophisticated candidate invariants or specifications from concrete test examples [21, 29, 30, 32–34]. Another application of machine-learning techniques is to encode soft constraints about program specifications in terms of a probabilistic model, and to infer a highly likely specification of a given program by performing a probabilistic inference on the model [2, 14, 16, 27]. In particular, Raychev et al.'s JSNice [27] uses a probabilistic model for describing type constraints and naming convention of JavaScript programs, which guides their cleaning process of messy JavaScript programs and is learnt from an existing codebase. Finally, machine-learning techniques have also been used to mine correct API usage from a large codebase and to synthesize code snippets using such APIs automatically [17, 28].

Our aim is different from those of the above works. We aim to improve a program analysis using machine-learning techniques, but our objective is not to find sophisticated invariants or specifications of a given program using these techniques. Rather it is to find a strategy for searching for such invariants. Notice that once this strategy is learnt automatically from an existing codebase, it is applied to multiple different programs. In the invariant-generation application, however, learning happens whenever a program is analyzed. Our work identifies a new challenging optimization problem related to learning such a strategy, and shows the benefits of Bayesian optimization for solving this problem.

Application of Bayesian Optimization. To the best of our knowledge, our work is the first application of Bayesian optimization to static program analysis. Bayesian optimization is a powerful optimization technique that has been successfully applied to solve a wide range of problems such as automatic algorithm configuration [13], hyperparameter optimization of machine-learning algorithms [36], planning, sensor placement, and intelligent user interface [3]. In this work, we use Bayesian optimization to find optimal parameters for adapting program analysis.

In our context, we believe Bayesian optimization is more powerful than other approaches such MCMC sampling used in, e.g., Reference [31], which relatively requires more evaluations of the (expensive) objective function. In this article, we showed that using Bayesian optimization outperforms other discrete optimization algorithms for our program-analysis application.

Abstraction Refinement. Applying the standard abstraction refinement to our problem is challenging and requires to solve yet another research problem. Most of the previous refinement-based work are for pointer analyses (e.g., Reference [39]). It is not straightforward to generalize their work to the program analyses considered in this work (e.g., flow-sensitive and/or context-sensitive interval analysis, widening with threshold).

9 CONCLUSION

In this article, we presented a novel approach for automatically learning a good strategy that adapts a static analysis to a given program. This strategy is learnt from an existing codebase efficiently via Bayesian optimisation, and it decides, for each program, which parts of the program should be treated with precise yet costly program-analysis techniques. This decision helps the analysis to strike a balance between cost and precision. Following our approach, we have implemented two variants of our buffer-overrun analyzer that adapt the degree of flow-sensitivity and context-sensitivity of the analysis. Our experiments confirm the benefits of Bayesian optimisation for learning adaptation strategies. They also show that the strategies learnt by our approach are highly effective: the cost of our variant analyses is comparable to that of flow- and context-*insensitive* analyses, while their precision is close to that of fully flow- and context-*sensitive* analyses.

As we already mentioned, our learning algorithm is nothing but a method for generalizing information from given programs to unseen ones. We believe that this cross-program generalization has a great potential for addressing open challenges in program analysis research, especially because the amount of publicly available source code (such as that in GitHub) has substantially increased. We hope that our results in this article give one evidence of this potential and get program-analysis researchers interested in this promising research direction.

REFERENCES

[1] T. Ball and S. Rajamani. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of the POPL*.

[2] Nels E. Beckman and Aditya V. Nori. 2011. Probabilistic, modular and scalable inference of typestate specifications. In *Proceedings of the PLDI*. 211–221.

[3] Eric Brochu, Vlad M. Cora, and Nando de Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR* abs/1012.2599. Retrieved from http://arxiv.org/abs/1012.2599.

[4] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. 2003. Modular verification of software components in C. In *Proceedings of the ICSE*.

[5] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the POPL*. 238–252.

[6] L. Dai. 1996. Convergence properties of ordinal comparison in the simulation of discrete event dynamic systems. *J. Optim. Theory Appl.* 91, 2 (1996), 363–388.

[7] S. Grebenshchikov, A. Gupta, N. Lopes, C. Popeea, and A. Rybalchenko. 2012. HSF(C): A software verifier based on horn clauses. In *Proceedings of the TACAS*.

[8] B. Gulavani, S. Chakraborty, A. Nori, and S. Rajamani. 2008. Automatically refining abstract interpretations. In *Proceedings of the TACAS*.

[9] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. 2013. From tests to proofs. *STTT* 15, 4 (2013), 291–303.

[10] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. 2004. Abstractions from proofs. In *Proceedings of the POPL*.

[11] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. 2003. Software verification with BLAST. In *Proceedings of the SPIN Workshop on Model Checking of Software*.

[12] Yu-Chi Ho. 1999. An explanation of ordinal optimization: Soft computing for hard problems. *Info. Sci.* 113, 34 (1999), 169–192. DOI: https://doi.org/10.1016/S0020-0255(98)10056-7

[13] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the LION*.

[14] Ted Kremenek, Andrew Y. Ng, and Dawson R. Engler. 2007. A factor graph model for software bug finding. In *Proceedings of the IJCAI*. 2510–2516.

[15] Percy Liang, Omer Tripp, and Mayur Naik. 2011. Learning minimal abstractions. In *Proceedings of the POPL*.

[16] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the PLDI*.

[17] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. In *Proceedings of the OOPSLA*. 997–1016.

[18] Jonas Mockus. 1994. Application of Bayesian approach to numerical methods of global and stochastic optimization. *J. Global Optim.* 4, 4 (1994). DOI: https://doi.org/10.1007/BF01099263

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

[20] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the*CC. Springer-Verlag, London, 213–228. Retrieved from http://dl.acm.org/citation.cfm?id=647478.727796.

[21] Aditya V. Nori and Rahul Sharma. 2013. Termination proofs from tests. In *Proceedings of the FSE*.

[22] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the PLDI*.

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

[24] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2014. Sparrow. Retrieved from http://ropas.snu.ac.kr/sparrow.

[25] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a strategy for adapting a program analysis via Bayesian optimisation. In *Proceedings of the OOPSLA*.

[26] Carl Edward Rasmussen and Christopher K. I. Williams. 2005. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

[27] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from "big code." In *Proceedings of the POPL*.

[28] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the PLDI*.

[29] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivancic, and Aarti Gupta. 2008. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the ISSTA*.

[30] Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. 2008. Mining library specifications using inductive logic programming. In *Proceedings of the ICSE*.

[31] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the*PLDI. ACM, New York, NY, 53–64. DOI: https://doi.org/10.1145/2594291.2594302

[32] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A data driven approach for algebraic loop invariants. In *Proceedings of the ESOP*.

[33] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. 2013. Verification as learning geometric concepts. In *Proceedings of the SAS*.

[34] Rahul Sharma, Aditya V. Nori, and Alex Aiken. 2012. Interpolants as classifiers. In *Proceedings of the CAV*.

[35] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the PLDI*.

[36] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. Practical Bayesian optimization of machine-learning algorithms. In *Proceedings of the NIPS*.

[37] Rainer Storn and Kenneth Price. 1997. Differential evolution—A simple and efficient heuristic for global optimization over continuous spaces. *J. Global Optim.* 11 (1997), 341–359. Issue 4.

[38] David J. Wales and Jonathan P. K. Doye. 1997. Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms. *J. Phys. Chem. A* 101, 28 (1997), 5111–5116.
[39] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in datalog. In *Proceedings of the PLDI*.
[40] Xin Zhang, Mayur Naik, and Hongseok Yang. 2013. Finding optimum abstractions in parametric dataflow analysis. In *Proceedings of the PLDI*.

Received March 2016; revised April 2017; accepted June 2017