저작자표시-비영리-변경금지 2.0 대한민국

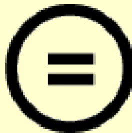
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학박사학위논문

예비분석과 기계학습을 이용하여 선별적으로 정확하게 정적분석을 하는 방법

Selectively Sensitive Static Analysis by Impact Pre-analysis and Machine Learning

2017년 8월

서울대학교 대학원

컴퓨터공학부

허 기 홍

예비분석과 기계학습을 이용하여 선별적으로 정확하게 정적분석을 하는 방법

지도교수 이 광 근

이 논문을 공학박사학위논문으로 제출함
2017년 5월

서울대학교 대학원
컴퓨터공학부
허 기 홍

허 기 홍의 박사학위논문을 인준함
2017년 7월

위 원 장	허 충 길	(인)
부위원장	이 광 근	(인)
위 원	김 건 희	(인)
위 원	류 석 영	(인)
위 원	오 학 주	(인)

Abstract

Selectively Sensitive Static Analysis
by Impact Pre-analysis and Machine Learning

Kihong Heo
Department of Computer Science and Engineering
College of Engineering
Seoul National University

In this dissertation, we present general methods to strike a right balance among soundness, precision, and scalability of static analysis. The key idea is to *selectively* apply precision-improving techniques, which risk being unscalable and unsound, only when they are likely to have benefits. We first propose a general principle to design an impact pre-analysis that estimates the impact of precision-improving techniques on the main analysis. Under the guidance of the pre-analysis result, the main analysis selectively turns on its sensitivity. We also propose machine-learning-based techniques that learn useful knowledge for the sensitivity from data generated by static analysis. With the data, we derive a classifier that quickly selects the effective sensitivity for the main analysis. We implemented these methods on top of a industrial-strength C static analyzer and the experimental result showed that this approach effectively balances precision and cost.

Keywords : Programming Language, Static Analysis, Selective Sensitivity, Pre-analysis, Machine Learning
Student Number : 2009-20919

Contents

1 Introduction **1**

1.1 Goal 1

1.2 Solution 2

1.3 Outline 4

2 Preliminaries **5**

2.1 Program 5

2.2 Collecting Semantics 6

2.3 Abstract Semantics 6

3 Selectively X-sensitive Analysis by Impact Pre-Analysis **9**

3.1 Introduction 9

3.2 Informal Description 11

3.3 Program Representation 17

3.4 Selective Context-Sensitive Analysis with Context-Sensitivity Parameter K 18

3.5 Impact Pre-Analysis for Finding K 22

3.5.1 Designing an Impact Pre-Analysis 22

3.5.2 Use of the Pre-Analysis Results 28

3.6 Application to Selective Relational Analysis 35

3.7 Experiments 40

3.8 Summary 42

4 Selectively X-sensitive analysis by learning data generated by impact pre-analysis 47
4.1 Introduction 47
4.2 Informal Explanation 50
4.2.1 Octagon Analysis with Variable Clustering 50
4.2.2 Automatic Learning of a Variable-Clustering Strategy 52
4.3 Octagon Analysis with Variable Clustering 56
4.3.1 Programs 56
4.3.2 Octagon Analysis 56
4.3.3 Variable Clustering and Partial Octagon Analysis 58
4.4 Learning a Strategy for Clustering Variables 59
4.4.1 Automatic Generation of Labeled Data 60
4.4.2 Features and Classifier 63
4.4.3 Strategy for Clustering Variables 64
4.5 Experiments 66
4.5.1 Effectiveness 67
4.5.2 Generalization 68
4.5.3 Feature Design 69
4.5.4 Choice of an Off-the-shelf Classification Algorithm 70
4.6 Summary 70

5 Selectively Unsound Analysis by Machine Learning 75
5.1 Introduction 75
5.2 Overview 78
5.2.1 Uniformly Unsound Analysis 78
5.2.2 Uniformly Sound Analysis 79
5.2.3 Selectively Unsound Analysis 80
5.2.4 Our Learning Approach 80
5.3 Our Technique 81
5.3.1 Parameterized Static Analysis 82
5.3.2 Learning a Classifier 83

5.4 Instance Analyses 87
5.4.1 A Generic, Selectively Unsound Static Analysis 87
5.4.2 Instantiation 1: Interval Analysis 91
5.4.3 Instantiation 2: Taint Analysis 91
5.5 Experiments 92
5.5.1 Setting 92
5.5.2 Effectiveness of Our Approach 93
5.5.3 Efficacy of OC-SVM 96
5.5.4 Feature Design 97
5.5.5 Time Cost 98
5.5.6 Discussion 98
5.6 Summary 100

6 Related Work **106**
6.1 Parametric Static Analysis 106
6.2 Goal-directed Static Analysis 107
6.3 Data-driven Static Analysis 108
6.4 Context-sensitivity and Relational Analysis 108
6.5 Unsoundness in Static Analysis 110

7 Conclusion **112**

List of Tables

3.1 Experimental Results of Selective Context-sensitivity 44
3.2 Experimental Results of Selective Relational Analysis 45

4.1 Features for Relations of Two Variables. 65
4.2 Characteristics of the Benchmark Programs 68
4.3 Experimental Results of Partial Octagon Analysis 72
4.4 Generalization Performance 73

5.1 Features for Typical Loops in C Programs 102
5.2 Features for Typical Library Calls in C Programs 103
5.3 The Number of Alarms in Interval Analysis 104
5.4 The Number of Alarms in Taint Analysis 105

List of Figures

3.1 Example Program 13
3.2 Example Context Selector 31

4.1 Example Program 51
4.2 Abstract Semantics of Primitive Commands in the Octagon Analysis 58
4.3 Abstract Semantics of Primitive Commands in the Impact Pre-analysis for the Octagon Analysis 62

5.1 Example Program 78
5.2 Static Analysis Selectively Unsound for Loops and Library Calls . . 88
5.3 Abstract Domain and Semantics for Interval Analysis 89
5.4 Abstract Domain and Semantics for Taint Analysis 90
5.5 Performance with Different Training and Test Data 95
5.6 Comparison between Selective, Binary, RandA, and RandB . 97

Chapter 1

Introduction

1.1 Goal

Static program analysis is a general method for automatically estimating the runtime behavior of programs before execution. Unlike testing (or dynamic analysis) which observes real program states by executing the target program with concrete inputs, static analysis *abstracts* possible program states without running the program. The abstraction may introduce false results that are not feasible in the real executions, but enables static analysis to handle infinitely many possible states and inputs for softwares. Hence, nowadays static analyzers are widely used for verification of safety-critical softwares, bug-detection, and compiler optimization.

The key factor of a static analyzer's performance is to choose a right abstraction of concrete semantics. Performance of static analysis is generally characterized by *soundness* (subsumption of concrete semantics), *precision* (prevention of false alarms), and *scalability* (efficiency for large programs). Because no static analysis can achieve these three properties in a single analyzer, traditional analyses have fallen into three categories with different abstractions that are suitable for the purposes: 1) sound, precise, yet unscalable analysis for verification, 2) sound, scalable, yet imprecise analysis for code optimization, 3) precise, scalable, yet unsound analysis for bug finding.

Traditionally, tuning the performance of these analyzers requires a large amount of tedious manual tasks to find a right abstraction. So far there have been many techniques that guide such abstraction a variety ways, such as context-sensitivity for function calls [48, 52], relational domain for variable relations [9, 30], and folklore heuristics for unsound analysis (e.g., destructing loops [12,22,58,60]), but their adaption strategies are mostly too simple-minded and uniform. When it comes to context-sensitivity, for example, the traditional k-callstring approach [48, 52] uniformly distinguishes their k-most recent call sites without any concern for their effectiveness. (e.g., uniform strategy with a fixed degree k). Therefore the resulting precise analysis easily sacrifices scalability and detectability without careful consideration.

In this dissertation, we propose general methods to automatically strike a right balance among soundness, precision, and scalability. The key principle is to *selectively* apply costly precision-improving techniques that risk being unscalable and unsound (e.g., context-sensitivity, relational domains, unsound strategies). Given a program to be analyzed, our methods selectively turn on such precision-improving techniques only when and where doing so is likely to improve precision without sacrificing detectability and scalability as far as possible.

1.2 Solution

The research problem is how to automatically select a right abstraction. The selection methods are enabled by the following techniques:

- **Selectively X-sensitive analysis by impact pre-analysis [37]:** We design a pre-analysis to estimate the impact of a precision-improving technique X on the main analysis's precision. Before the main analysis starts, the impact pre-analysis analyzes a given target program and selects a cost-effective X-sensitivity. With the guidance of the impact pre-analysis, the main analysis applies the X-sensitivity only when it is likely to improve the precision. We formalize this approach and prove that the analysis always benefits from the

pre-analysis results. We implemented two instances (context-sensitivity and relational analysis) in an industrial-strength interval analyzer for full C and experimentally showed the effectiveness.

- **Selectively X-sensitive analysis by learning data generated by impact pre-analysis [18]:** We design a method for automatically learning an effective strategy for X-sensitivity. For some precision-improving techniques (e.g., relational analysis), the impact pre-analyses are still too costly, so infeasible with larger programs. Instead of using the pre-analysis as an *online estimator*, we employ it as an *offline teacher*. The pre-analysis automatically labels training data in the codebase as positive or negative. Then, we derive a binary classifier using an off-the-shelf machine learning algorithm. Given a target program, this classifier quickly estimates the selective X-sensitivity. We implemented a selectively relational analysis with this method on top of a static buffer-overflow detector for C programs and showed the effectiveness.

- **Selectively unsound analysis by machine learning [19]**: We present a machine-learning-based technique for selectively applying unsoundness. Existing bug-finding static analyzers are unsound in order to be precise and scalable in practice. However, they are uniformly unsound and hence at the risk of missing a large amount of real bugs. By being sound, we can improve the detectability of the analyzer but it often suffers from a large number of false alarms. Our approach aims to strike a balance between these two approaches by selectively allowing unsoundness only when it is likely to reduce false alarms, while retaining true alarms. We use a machine learning technique to learn such harmless unsoundness. We implemented our technique in two static analyzers for full C. One is for a taint analysis for detecting format-string vulnerabilities, and the other is for an interval analysis for buffer-overflow detection. The experimental results show that our approach significantly improves the detectability of the original unsound analysis without sacrificing the precision.

1.3 Outline

The rest of this dissertation is organized by as follows:

- Chapter 3 presents the selectively X-sensitive analysis by impact pre-analysis.
- Chapter 4 presents the selectively X-sensitive analysis by learning the results of impact pre-analysis.
- Chapter 5 presents the selectively unsound analysis by machine learning.
- Chapter 6 discusses related works and Chapter 7 concludes the dissertation.

Chapter 2

Preliminaries

2.1 Program

We assume that a program P is represented by a control flow graph $(\mathbb{C}, \to, \mathbb{F}, \iota)$ where $\mathbb{C}$ is the finite set of nodes, $(\to) \subseteq \mathbb{C} \times \mathbb{C}$ denotes the control flow relation between nodes, $\mathbb{F}$ is the set of procedure ids, and $\iota \in \mathbb{C}$ is the entry node of the `main` procedure. The entry node ι does not have predecessors. Each node $c \in \mathbb{C}$ has program command $\mathtt{cmd}(c)$ that is one of the followings:

$$lv := e \mid lv := \mathtt{alloc}_l(e) \mid \mathtt{assume}(x < e)$$

where l-value expression (lv) and expression (e) are defined as follows:

$$\begin{array}{lrcl} \text{expression} & e & \to & n \mid e{+}e \mid lv \mid \&lv \\ \text{l-value} & lv & \to & x \mid *e \mid e[e] \end{array}$$

An expression is a numeric value (n), a binary operation of two expressions ($e{+}e$), a l-value expression (lv), or an address of a l-value expression ($\&lv$). A l-value expression is a variable (x), a dereference of an expression ($*e$), or an array access ($e[e]$). Command $lv := e$ assigns the value of e to the location of lv. Command $lv := \mathtt{alloc}_l(e)$ allocates an array of which size is e at allocation site l. Command $\mathtt{assume}(x < e)$ limits the program executions that invalidate condition $x < e$.

2.2 Collecting Semantics

The concrete domain $\mathbb{D}$ represents a set of reachable states at each control point:

$$\mathbb{D} = \mathbb{C} \to 2^{\mathbb{S}}$$

The domain of state $\mathbb{S}$ is defined as follows:

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

where $\mathbb{L}$ is a set of concrete locations and $\mathbb{V}$ is a set of concrete values. The collecting semantics of a program is characterized by the least fixpoint of semantic function $F \in \mathbb{D} \to \mathbb{D}$:

$$F(X) = \lambda c \in \mathbb{C}.f_c\big(\bigcup_{c' \to c} X(c')\big)$$

where $f_c \in 2^{\mathbb{D}} \to 2^{\mathbb{D}}$ is a semantic function for program point c. We follow the standard definition of the concrete semantic function.

2.3 Abstract Semantics

We abstract the collecting semantics by the following Galois connection [8]:

$$\mathbb{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{\mathbb{D}}$$

where the domain $\hat{\mathbb{D}}$ is a set of maps from program points ($\mathbb{C}$) to abstract states ($\mathbb{S}$):

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

The Galois connection of (α, γ) is defined as pointwise lifting of the Galois connection of $(\alpha_{\mathbb{S}}, \gamma_{\mathbb{S}})$:

$$2^{\mathbb{S}} \underset{\alpha_{\mathbb{S}}}{\overset{\gamma_{\mathbb{S}}}{\leftrightarrows}} \hat{\mathbb{S}}$$

The abstract state is a set of maps from abstract locations ($\hat{\mathbb{L}}$) to abstract values ($\mathbb{V}$):

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

where

$$\begin{array}{rcl} \hat{\mathbb{L}} & = & \mathit{Var} + \mathit{Allocsite} \\ \hat{\mathbb{V}} & = & \hat{\mathbb{Z}} \times 2^{\hat{\mathbb{L}}} \times 2^{\mathit{Allocsite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}} \\ \hat{\mathbb{Z}} & = & \{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, +\infty\} \ \wedge \ l \leq u\} \cup \{\bot\} \end{array}$$

An abstract location is a variable (Var) or an allocation site ($\mathit{Allocsite}$). We abstract all elements of an array as its allocation site l. An abstract value is a numeric value ($\hat{\mathbb{Z}}$), points-to information ($2^{\hat{\mathbb{L}}}$), or a set of abstract array blocks ($2^{\mathit{Allocsite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}}$). A set of numeric values is abstracted as an interval. We collect a set of pointees for the point-to information. An abstract array block is a triple of its allocation site, offset, and size.

The abstract semantics is characterized as a least fixpoint of abstract semantic function $\hat{F} \in \hat{\mathbb{D}} \to \hat{\mathbb{D}}$:

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c\big(\bigsqcup_{c' \to c} \hat{X}(c')\big)$$

where $\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is a abstract semantic function for program point c. The abstract semantic function is defined as follows:

$$\hat{f}_c(\hat{s}) = \begin{cases} \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \hat{\mathcal{V}}(e)(\hat{s})] & \mathsf{cmd}(c) = lv{:=}e \\ \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \langle \bot, \bot, \{\langle l, [0,0], \hat{\mathcal{V}}(e)(\hat{s}).1\rangle\}\rangle] & \mathsf{cmd}(c) = lv \ := \ \mathtt{alloc}_l(e) \\ \hat{s}[x \mapsto \langle \hat{s}(x).1 \sqcap [-\infty, \mathsf{u}\big(\hat{\mathcal{V}}(e)(\hat{s}).1\big)], \hat{s}(x).2, \hat{s}(x).3\rangle] & \mathsf{cmd}(c) = \mathtt{assume}(x < e) \end{cases}$$

$\overset{w}{\mapsto}$ denotes the weak update operator[1]:

$$\hat{s}[\{l_1, \cdots, l_n\} \overset{w}{\mapsto} v] = \hat{s}[l_1 \mapsto \hat{s}(l_1) \sqcup v, \cdots, l_n \mapsto \hat{s}(l_n) \sqcup v]$$

[1]For brevity, we only consider weak updates. Our implementation supports strong updates and the extension is straightforward.

Auxiliary functions $\hat{\mathcal{V}}(e)(\hat{s})$ and $\hat{\mathcal{L}}(e)(\hat{s})$ compute abstract values and locations for each expression and l-value expression.

$$\begin{array}{rcl}
\hat{\mathcal{V}}(e) & \in & \hat{\mathbb{S}} \to \hat{\mathbb{V}} \\
\hat{\mathcal{V}}(n)(\hat{s}) & = & \langle [n,n], \bot, \bot \rangle \\
\hat{\mathcal{V}}(e_1 + e_2)(\hat{s}) & = & \hat{\mathcal{V}}(e_1)(\hat{s}) \hat{+} \hat{\mathcal{V}}(e_2)(\hat{s}) \\
\hat{\mathcal{V}}(lv)(\hat{s}) & = & \bigsqcup \{ \hat{s}(l) \mid l \in \hat{\mathcal{L}}(lv)(\hat{s}) \} \\
\hat{\mathcal{V}}(\&lv)(\hat{s}) & = & \langle \bot, \hat{\mathcal{L}}(lv)(\hat{s}), \bot \rangle
\end{array}$$

$\hat{\mathcal{V}}$ computes abstract values for each expression. Integer n evaluates to the corresponding interval $[n, n]$. For binary operations, we compute the abstract version of the operations on inductively evaluated abstract values by $\hat{\mathcal{V}}$. We skip the conventional definition of the abstract binary ($\hat{+}$) and join ($\sqcup$) operations. For l-value expressions, we join all abstract values that associated with the abstract locations for the l-value expressions. An address of a l-value expression evaluates to the all abstract locations of the l-value expression.

$$\begin{array}{rcl}
\hat{\mathcal{L}}(lv) & \in & \hat{\mathbb{S}} \to 2^{\hat{\mathbb{L}}} \\
\hat{\mathcal{L}}(x)(\hat{s}) & = & \{x\} \\
\hat{\mathcal{L}}(*e)(\hat{s}) & = & \hat{\mathcal{V}}(e)(\hat{s}).2 \cup \{ l \mid \langle l, _, _ \rangle \in \hat{\mathcal{V}}(e)(\hat{s}).3 \} \\
\hat{\mathcal{L}}(e_1[e_2])(\hat{s}) & = & \{ l \mid \langle l, _, _ \rangle \in \hat{\mathcal{V}}(e_1)(\hat{s}).3 \}
\end{array}$$

The abstract location of a variable (x) is the variable itself. A dereference of an expression ($*e$) denotes all the abstract locations involved in the expression: all the abstract locations in the points-to set and the set of array blocks. An array access ($e_1[e_2]$) denotes a set of all allocation sites of the array blocks evaluated from the array expression (e_1).

Lemma 1 (Soundness) *If $\hat{F}$ is a sound approximation of F then the abstract semantics $\mathsf{lfp}\hat{F}$ is a sound approximation of the concrete semantics $\mathsf{lfp}F$, i.e.,*

$$\alpha \circ F \sqsubseteq \hat{F} \circ \alpha \implies \alpha(\mathsf{lfp}F) \sqsubseteq \mathsf{lfp}\hat{F}$$

Chapter 3

Selectively X-sensitive Analysis by Impact Pre-Analysis

3.1 Introduction

Handling procedure calls in static analysis with a right balance between precision and cost is challenging. To precisely analyze procedure calls and returns, the analysis has to distinguish calls to the same procedure by their different calling contexts. However, a simple-minded, uniform context-sensitivity at all call sites easily makes the resulting analysis non cost-effective. For example, imagine a program analysis for proving the safety of array accesses that uses the k-callstring approach [48, 52] for abstracting calling contexts. The k-callstring approach distinguishes two calls to the same procedure whenever their k-most recent call sites are different. To make this context-sensitive analysis cost-effective, we need to tune the k values at the call sites in a way that we should increase the k value only where the increased precision contributes to the proof of array-access safety. If we simply use the same fixed k for all the call sites, the analysis would end up becoming either unnecessarily precise and costly, or not precise enough to prove the safety of many array accesses.

In this paper, we present a method for performing *selective context-sensitive* analysis, which applies the context-sensitivity only when and where doing so is likely to improve the precision that matters for the analysis's ultimate goal. Our method consists of two steps. The first step is a pre-analysis that estimates the behavior of the main analysis under the full context-sensitivity (i.e. using ∞-callstrings). The pre-analysis focuses only on estimating the impact of context-sensitivity on the main analysis. Hence, it aggressively abstracts the other semantic aspects of the main analysis. The second step is the main analysis with selective context-sensitivity. This analysis uses the results of the pre-analysis, selects influential call sites for precision, and selectively applies context-sensitivity only to these call sites. Our method can be instantiated with a range of static analyses, and provides a guideline for designing impact pre-analyses for them, in particular, an efficient way of implementing those pre-analyses based on graph reachability.

One important feature of our method is that the pre-analysis-guided context-sensitivity pays off at the subsequent selective context-sensitive analysis. One way to see the subtlety of this impact realization is to note that the pre-analysis and the selective main analysis are incomparable in precision: the pre-analysis is more precise than the main analysis in terms of context sensitivity, but it is worse than the main analysis in tracking individual program statements. Despite this mismatch, our guidelines for designing an impact pre-analysis and the resulting selective context-sensitivity ensure that the selective context-sensitive main analysis is at least as precise as the fully context-sensitive pre-analysis, as far as given queries are concerned.

We have implemented our method on an existing industrial-strength interval analyzer for full C. The method led to the reduction of alarms from 6.6 to 48.3%, with average 24.4%, compared with the baseline context-insensitive analysis, while increasing the analysis cost from 9.4 to 50.5%, with average 27.8%.

The general principle behind the design and the use of our impact pre-analysis can be used for developing other types of selective analyses. We show its applicability by following the same principle and developing a selective relational analysis

that keeps track of relationships between variables, only when tracking them are likely to help the main analysis answer given queries. In this case, the impact pre-analysis is fully relational while it aggressively abstracts other semantic aspects. The experiments show that our selective relational analysis achieves competitive cost-precision tradeoffs when applied to real-world benchmark programs.

We summarize the contributions of the paper:

- We present a method for performing selective context-sensitive analysis that receives guidance from an impact pre-analysis.
- We show that the general idea behind our selective method is not limited to context-sensitivity. We present a selective relational analysis that is guided by an impact pre-analysis.
- We experimentally show the effectiveness of selective analyses designed according to our method, with real-world C programs.

3.2 Informal Description

We illustrate our approach using the interval domain and the program in Figure 3.1, which is adopted from `make-3.76.1`. Procedure `xmalloc` is a wrapper of the `malloc` function. It is called twice in procedure `multi_glob`, once with the argument `size` (line 4) and again with an input from the environment (line 6). The `main` routine of this program calls procedure `f` and `g`. Procedure `multi_glob` is called in `f` and `g` with different argument values.

The program contains two queries. The first query at line 5 asks whether `p` points to a buffer of size larger than 1. The other query at line 7 asks a similar question, but this time for the pointer variable `q`. Note that the first query always holds, but the second query is not necessarily true.

Context-insensitive analysis If we analyze the program using a context-insensitive interval analysis, we cannot prove the first query. Since the analysis is insensitive

to calling contexts, it estimates the effect of `xmalloc` under all the possible inputs, and uses this same estimation as the result of every call. Note that an input to `xmalloc` at line 6 can be any integer, and the analysis concludes that `xmalloc` allocates a buffer of size in $[-\infty, +\infty]$.

Context-sensitive analysis A natural way to fix this precision issue is to increase the context-sensitivity. One popular approach is k-CFA analysis [48, 52]. It uses sequences of call sites up to length k to distinguish calling contexts of a procedure, and analyzes the procedure separately for such distinguished calling contexts. For instance, 3-CFA analyzes the procedure `xmalloc` separately for each of the following calling contexts:

$$\begin{array}{cccc} 4 \cdot 10 \cdot 14 & 4 \cdot 10 \cdot 15 & 4 \cdot 11 \cdot 16 & 4 \cdot 11 \cdot 17 \\ 6 \cdot 10 \cdot 14 & 6 \cdot 10 \cdot 15 & 6 \cdot 11 \cdot 16 & 6 \cdot 11 \cdot 17 \end{array} \tag{3.1}$$

Here $a \cdot b \cdot c$ denotes a sequence of call sites a, b and c (we use the line numbers as call sites), with a being the most recent call. Note that the 3-CFA analysis can prove the first query: the analysis analyzes the first four contexts separately and infers that a buffer of size greater than 1 gets allocated under these calling contexts.

Need of selective context-sensitivity However, using such a "uniform" context-sensitivity is not ideal. It is often too expensive to run such an analysis with high enough k, such as $k \geq 3$ that our example needs. More importantly, for many procedure calls, increasing context-sensitivity does not help—either it does not improve the analysis results of these calls, or the increased precision is not useful for answering queries. For instance, at the second query, for every $k \geq 0$, the k-CFA analysis concludes that `p` points to a buffer of size $[-\infty, +\infty]$. Also, it is unnecessary to analyze `g` separately for call sites 16 and 17, because those two calls have the same effect on the query.

```
char* xmalloc (int n) { return malloc(n); }

void multi_glob (int size) {
  p = xmalloc (size);
  assert (sizeof(p) > 1);     // Query 1
  q = xmalloc (input());
  assert (sizeof(q) > 1);     // Query 2
}

void f (int x) { multi_glob (x); }
void g ()      { multi_glob (4); }

int main() {
  f (8);
  f (16);
  g ();
  g ();
}
```

Figure 3.1: Example Program

Our selective context-sensitivity With our approach, an analysis can analyze procedures with only needed context-sensitivity. It analyzes a procedure separately for a calling context if doing so is likely to improve the precision of the analysis and reduce false alarms in its answers for given queries. For the example program, our analysis first predicts that increasing context-sensitivity is unlikely to help answer the second query (line 7) accurately, but is likely to do so for the first query (line 5). Next, the analysis finds out that we can bring the full benefit of context-sensitivity for the first query, by distinguishing only the following four types of calling contexts of `xmalloc`:

$$4 \cdot 10 \cdot 14, \quad 4 \cdot 10 \cdot 15, \quad 4 \cdot 11, \quad \text{all the other contexts} \tag{3.2}$$

Note that contexts 4·11·16 and 4·11·17 are merged into a single context 4·11. This merging happens because the analysis figures out that two callers of `g` (line 16 and 17) do not provide any useful information for resolving the first query. Finally, the analysis analyzes the given program using the interval domain while distinguishing calling contexts above and their suffixes (i.e., 10 · 14, 10 · 15, 14, 15, 11). This selective context-sensitive analysis is able to prove the first query.

Impact pre-analysis Our key idea is to approximate the main analysis under full context-sensitivity using a pre-analysis, and estimate the impact of context-sensitivity on the results of the main analysis. This impact pre-analysis uses a simple abstract domain and transfer functions, and can be run efficiently even with full context-sensitivity.

For instance, we approximate the interval analysis in this example using a pre-analysis with two abstract values: ★ and ⊤. Here ⊤ means all intervals, and ★ intervals of the form $[l, u]$ with $0 \leq l \leq u$. A typical abstract state in this domain is $[x : \top, y : \bigstar]$, which means the following set of states in the interval domain:

$$\{[x : [l_x, u_x], y : [l_y, u_y]] \mid l_x \leq u_x \wedge 0 \leq l_y \leq u_y\}.$$

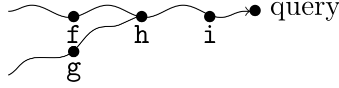
This simple abstract domain of the pre-analysis is chosen because we are interested in showing the absence of buffer overruns and the analysis proves such properties only when it finds non-negative intervals for buffer sizes and indices.

We run this pre-analysis under full context-sensitivity (i.e., ∞-CFA). For our example program, we obtain a summary of the procedure `xmalloc` with eight entries, each corresponding to a different context in (3.1). The third column of the table below shows this summary:

Contexts	Size of the allocated buffer in `xmalloc`	
	Main analysis	Pre-analysis
$4 \cdot 10 \cdot 14$	$[8, 8]$	$\bigstar$
$4 \cdot 10 \cdot 15$	$[16, 16]$	$\bigstar$
$4 \cdot 11 \cdot 16$	$[4, 4]$	$\bigstar$
$4 \cdot 11 \cdot 17$	$[4, 4]$	$\bigstar$
$6 \cdot 10 \cdot 14$	$[-\infty, +\infty]$	$\top$
$6 \cdot 10 \cdot 15$	$[-\infty, +\infty]$	$\top$
$6 \cdot 11 \cdot 16$	$[-\infty, +\infty]$	$\top$
$6 \cdot 11 \cdot 17$	$[-\infty, +\infty]$	$\top$

The second column of the table shows the results of the interval analysis with full context-sensitivity. Note that the pre-analysis in this case precisely estimates the impact of context-sensitivity: it identifies calling contexts (i.e., the first four contexts in the table) where the interval analysis accurately tracks the size of the allocated buffer in `xmalloc` under the full context-sensitivity. In general, our pre-analysis might lose precision and use $\top$ more often than in the ideal case. However, even when such approximation occurs, it does so only in a sound manner—if the pre-analysis computes $\bigstar$ for the size of a buffer, the interval analysis under full context-sensitivity is guaranteed to compute a non-negative interval.

Use of pre-analysis results Next, from the pre-analysis results, we select calling contexts that help improve the precision regarding given queries. We first identify queries whose expressions are assigned with $\bigstar$ in the pre-analysis run. In our example, the pre-analysis assigns $\bigstar$ to the expression `sizeof(p)` in the first query. We regard this as a good indication that the interval analysis under full context-sensitivity is likely to estimate the value of `sizeof(p)` accurately. Then, for each query that is judged promising, we consider the slice of the program that contributes to the query. We conclude that all the calls made in the slice should be tracked precisely. For example, if a slice for a query looks as follows:



Then, we derive calling contexts f, g, {h·f, h·g}, and {i·h·f, i·h·g} for procedure f, g, h, and i, respectively. However, if the slice involves a recursive call, we exclude the query since otherwise, we need infinitely many different calling contexts. In our example program, the slice for the first query includes all the execution paths from lines 11, 14, and 15 to line 5. Note that call-sites 16 and 17 are not included in the slice, and that all the calling contexts of xmalloc in this slice are: $4 \cdot 10 \cdot 14$, $4 \cdot 10 \cdot 15$, and $4 \cdot 11$. Our analysis decides to distinguish these contexts and their suffixes.

Impact realization Our method guarantees that the impact estimation under full context-sensitivity pays off at the subsequent selective context-sensitive analysis. That is, in our example program, the selective main analysis, which distinguishes only the contexts in (3.2), is guaranteed to assign a nonnegative interval to the expression sizeof(p) at the first query. This guarantee holds because our selective context-sensitive analysis distinguishes all the calling contexts that matter for the selected queries (Section 3.5.2) and ensures that undistinguished contexts are isolated from the distinguished contexts (Section 3.4). For instance, although the call to xmalloc at line 6 is analyzed in a context-insensitive way, our analysis ensures that xmalloc in this case returns only to line 6, not to line 4.

Application to relational analysis Behind our approach lies a general principle for developing a static analysis that selectively uses precision-improving techniques, such as context-sensitivity and relational abstract domains. The principle is to develop an impact pre-analysis that finds out when and where the main static analysis under the full precision setting is likely to have an accurate result, and to choose an appropriate precision setting of the main analysis based on the results of this pre-analysis.

For instance, suppose that we want to develop a selective version of the octagon analysis, which tracks only some relationships between program variables that are likely to be tracked well by the octagon analysis and also to help the proofs of given queries. To achieve this goal, we design an impact pre-analysis that aims at finding when and where the original octagon analysis is likely to infer precise relationships between program variables. In Section 3.6, we describe this selective octagon analysis in detail.

3.3 Program Representation

A node $c \in \mathbb{C}$ in the program is one of the five types:

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

Each procedure $f \in \mathbb{F}$ has one entry node and one exit node. Given a node $c \in \mathbb{C}$, $\mathsf{fid}(c)$ denotes the procedure enclosing the node. Each call-site in the program is represented by a pair of call and return nodes. Given a return node $c \in \mathbb{C}_r$, we write $\mathsf{callof}(c)$ for the corresponding call node. We denote the set of call edges by $\rightarrowtail$:

$$(\rightarrowtail) = \{(c_1, c_2) \mid c_1 \rightarrow c_2 \ \wedge \ c_1 \in \mathbb{C}_c \ \wedge \ c_2 \in \mathbb{C}_e\}$$

and the set of return edges by $\dashrightarrow$:

$$(\dashrightarrow) = \{(c_1, c_2) \mid c_1 \rightarrow c_2 \ \wedge \ c_1 \in \mathbb{C}_x \ \wedge \ c_2 \in \mathbb{C}_r\}.$$

We assume for simplicity that there are no indirect function calls such as calls via function pointers.

We associate a primitive command with each node c of our control flow graph, and denote it by $\mathsf{cmd}(c)$. For brevity, we consider simple primitive commands specified by the following grammar:

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

where e is an arithmetic expression: $e \rightarrow n \mid x \mid e + e \mid e - e$. We denote the set of all program variables by Var.

For simplicity, we handle parameter passing and return values of procedures via simple syntactic encoding. Recall that we represent a call statement $x := f_p(e)$ (where p is a formal parameter of procedure f) with call and return nodes. In our program, the call node has command $p := e$, so that the actual parameter e is assigned to the formal parameter p. For return values, we assume that each procedure f has a variable r_f and the return value is assigned to r_f: that is, we represent return statement `return` e of procedure f by $r_f := e$. The return node has command $x := r_f$, so that the return value is assigned to the original return variable. We assume that there are no global variables in the program, all parameters and local variables of procedures are distinct, and there are no recursive procedures.

3.4 Selective Context-Sensitive Analysis with Context-Sensitivity Parameter K

We consider selective context-sensitive analyses specified by the following data: (1) a domain $\mathbb{S}$ of abstract states, which forms a complete lattice structure $(\mathbb{S}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$; (2) an initial abstract state $s_I \in \mathbb{S}$ at the entry of the `main` procedure; (3) a monotone abstract semantics of primitive commands $[\![cmd]\!] : \mathbb{S} \rightarrow \mathbb{S}$; (4) a context selector K that maps procedures to sets of calling contexts (sequences of call nodes):

$$K \in \mathbb{F} \rightarrow \wp(\mathbb{C}_c^*).$$

For each procedure f, the set $K(f)$ specifies calling contexts that the analysis should differentiate while analyzing the procedure. We sometimes abuse the notation and denote by K the entire set of calling contexts in K: we write $\kappa \in K$ for $\kappa \in \bigcup_{f \in \mathbb{F}} K(f)$.

With the above data, we design a selective context-sensitive analysis as follows. First, we differentiate nodes with contexts in K, and define a set $\mathbb{C}_K \subseteq \mathbb{C} \times \mathbb{C}_c^*$ of context-enriched nodes:

$$\mathbb{C}_K = \{(c, \kappa) \mid c \in \mathbb{C} \ \wedge \ \kappa \in K(\mathsf{fid}(c))\}.$$

The control flow relation $(\rightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ is extended to $\rightarrow_K$ on $\mathbb{C}_K$:

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

$$(c, \kappa) \rightarrow_K (c', \kappa') \quad \textit{iff}$$
$$\begin{cases} c \rightarrow c' \ \wedge \ \kappa' = \kappa & (c' \notin \mathbb{C}_e \uplus \mathbb{C}_r) \\ c \rightarrow c' \ \wedge \ \kappa' = c ::_K \kappa & (c \in \mathbb{C}_c \ \wedge \ c' \in \mathbb{C}_e) \\ c \rightarrow c' \ \wedge \ \kappa = \mathsf{callof}(c') ::_K \kappa' & (c \in \mathbb{C}_x \ \wedge \ c' \in \mathbb{C}_r) \end{cases}$$

where $(::_K) \in \mathbb{C}_c \times \mathbb{C}_c^* \rightarrow \mathbb{C}_c^*$ *updates contexts according to* K*:*

$$c ::_K \kappa = \begin{cases} c \cdot \kappa & (c \cdot \kappa \in K) \\ \epsilon & \textit{otherwise} \end{cases}$$

where ϵ *is the empty call sequence.*

In our analysis, ϵ is used to represent *all the other contexts* not included in K, and we assume that K includes ϵ if it is necessary. For instance, consider a program where f has three different calling contexts κ_1, κ_2, and κ_3. When the analysis differentiates κ_1 only, undistinguished contexts κ_2 and κ_3 are represented by ϵ. Thus, $K(f) = \{\kappa_1, \epsilon\}$. Note that our analysis isolates undistinguished contexts from distinguished ones: ϵ means only κ_2 or κ_3, not κ_1.

Example 1 *The analysis is context-insensitive when* $K = \lambda f.\{\epsilon\}$ *and fully context-sensitive when* $K = \lambda f.\mathbb{C}_c^*$*. Our selective context-sensitive analysis in Section 3.2 uses the following context selector* $K = \{\texttt{main} \mapsto \{\epsilon\}, \texttt{f} \mapsto \{14, 15\}, \texttt{g} \mapsto \{\epsilon\}, \texttt{multi_glob} \mapsto \{10 \cdot 14, 10 \cdot 15, 11\}, \texttt{xmalloc} \mapsto \{4 \cdot 10 \cdot 14, 4 \cdot 10 \cdot 15, 4 \cdot 11, \epsilon\}\}$*.*

Next, we define the abstract domain $\mathbb{D}$ of the analysis:

$$\mathbb{D} = (\mathbb{C}_K \to \mathbb{S}) \tag{3.3}$$

The analysis keeps multiple abstract states at each program node c, one for each context $\kappa \in K(\mathsf{fid}(c))$. The abstract transfer function F of the analysis works on $\mathbb{C}_K$, and it is defined as follows:

$$F(X)(c,\kappa) = [\![\mathsf{cmd}(c)]\!]\Big(\bigsqcup_{(c_0,\kappa_0) \to_K (c,\kappa)} X(c_0,\kappa_0)\Big). \tag{3.4}$$

The static analysis computes an abstract element $X \in \mathbb{D}$ satisfying the following condition:

$$s_I \sqsubseteq X(\iota, \epsilon) \ \wedge\ \forall (c,\kappa) \in \mathbb{C}_K.\ F(X)(c,\kappa) \sqsubseteq X(c,\kappa) \tag{3.5}$$

In general, many X can satisfy the condition in (3.5). Some analyses compute the least X satisfying (3.5). Other analyses use a widening operator [8], $\bigtriangledown : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, and compute not necessarily the least, but some solution of (3.5).

Example 2 (Interval Analysis) *The interval analysis is a standard example that uses a widening operator. Let* $\mathbb{I}$ *be the domain of intervals:* $\mathbb{I} = \{[l,u] \mid l,u \in \mathbb{Z} \cup \{-\infty, +\infty\} \wedge l \le u\}$*. Using this domain, we specify the rest of the analysis:*

1. *The abstract states are* $\bot$ *or functions from program variables to their interval values:* $\mathbb{S} = \{\bot\} \cup (\mathsf{Var} \to \mathbb{I})$
2. *The initial abstract state is:* $s_I(x) = [-\infty, +\infty]$*.*

3. *The abstract semantics of primitive commands is:*

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

where $[\![e]\!]$ *is the abstract evaluation of the expression* e*:*

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

4. *The last component of the analysis is a widening operator, which is defined as a pointwise lifting of the following widening operators* $\nabla_I : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ *for intervals:*

$$\begin{array}{rcl} [l, u] \nabla_I [l', u'] & = & [\mathrm{ite}(l' < l, \mathrm{ite}(l' < 0, -\infty, 0), l), \\ & & \mathrm{ite}(u' > u, +\infty, u)] \end{array}$$

where $\mathrm{ite}(p, a, b)$ *evaluates to* a *if* p *is true and* b *otherwise. The above widening operator uses* 0 *as a threshold, which is useful when proving the absence of buffer overruns.*

Queries Queries are triples in $\mathcal{Q} \subseteq \mathbb{C} \times \mathbb{S} \times \mathsf{Var}$, and they are given as input to our static analysis. A query (c, s, x) represents an assertion that every reachable concrete state at node c is over-approximated by the abstract state s. The last component x describes that the query is concerned with the value of the variable x. For instance, in the interval analysis, a typical query is

$$(c, \;\; \lambda y.\, \mathrm{if}\ (y = x)\ \mathrm{then}\ [0, \infty]\ \mathrm{else}\ \top, \;\; x)$$

for some variable x. It asserts that at program node c, the variable x should always have a non-negative value. Proving the queries or identifying those that are likely to be violated is the goal of the analysis.

3.5 Impact Pre-Analysis for Finding K

Suppose that we would like to develop a selective context-sensitive analysis in Section 3.4 for a given program and given queries, using one of the existing abstract domains specified by the following data:

$$(\mathbb{S},\ \ s_I \in \mathbb{S},\ \ [\![-]\!] : \mathbb{S} \to \mathbb{S}),$$

To achieve our aim, we need to construct K a specification on context-sensitivity for the given program and queries. Once this construction is done, the rest is standard. The analysis can analyze the program under partial context-sensitivity, using the induced abstract domain $\mathbb{D}$ and transfer function $F : \mathbb{D} \to \mathbb{D}$ for this program in (3.3) and (3.4). We assume that the analysis employs the fixpoint algorithm based on widening operation $\bigtriangledown : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$.

How should we automatically choose an effective K that balances the precision and cost of the induced interprocedural analysis? In this section, we give an answer to this question. In Section 3.5.1, we present an impact pre-analysis, which estimates the behavior of the main analysis $(\mathbb{S}, s_I, [\![-]\!])$ under full context-sensitivity. In Section 3.5.2, we describe how to use the results of this pre-analysis for constructing an effective context selector K. Throughout the section, we fix our main analysis to $(\mathbb{S}, s_I, [\![-]\!])$.

3.5.1 Designing an Impact Pre-Analysis

An impact pre-analysis for context sensitivity aims at estimating the main analysis $(\mathbb{S}, s_I, [\![-]\!])$ under full context-sensitivity. It is specified by the following data:

$$(\mathbb{S}^\sharp,\ \ s_I^\sharp \in \mathbb{S}^\sharp,\ \ [\![-]\!]^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp,\ \ K_\infty).$$

This specification and the way that the data are used in our pre-analysis are fairly standard. $\mathbb{S}^\sharp$ and $[\![cmd]\!]^\sharp$ are, respectively, the domain of abstract states and the abstract semantics of cmd used by the pre-analysis, and $s_I^\sharp$ is an initial state.

$K_\infty = \lambda f.\mathbb{C}_c^*$ is the context selector for full context-sensitivity. The pre-analysis uses the abstract domain $\mathbb{D}^\sharp = \mathbb{C}_{K_\infty} \to \mathbb{S}^\sharp$ and the following transfer function $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ for the given program:

$$F^\sharp(X)(c,\kappa) = [\![\mathsf{cmd}(c)]\!]^\sharp(\bigsqcup_{(c_0,\kappa_0)\to_{K_\infty}(c,\kappa)} X(c_0,\kappa_0)).$$

It computes the least X satisfying

$$s_I^\sharp \sqsubseteq X(\iota,\epsilon) \ \wedge \ \forall(c,\kappa) \in \mathbb{C}_K.\ F^\sharp(X)(c,\kappa) \sqsubseteq X(c,\kappa) \tag{3.6}$$

What is less standard is the soundness and efficiency conditions for our pre-analysis, which provides a guideline on the design of these pre-analyses. Let us discuss these conditions separately.

Soundness condition Intuitively, our soundness condition says that all the components of the pre-analysis have to over-approximate the corresponding ones of the main analysis.[1] This is identical to the standard soundness requirement of a static program analysis, except that the condition is stated not over the concrete semantics of a given program, but over the main analysis. The condition has the following four requirements:

1. There should be a concretization function $\gamma : \mathbb{S}^\sharp \to \wp(\mathbb{S})$. This function formalizes the fact that an abstract state of the pre-analysis means a set of abstract states of the main analysis.

2. The initial abstract state of the pre-analysis has to overapproximate the initial state of the main analysis, i.e., $s_I \in \gamma(s_I^\sharp)$.

[1] We design a pre-analysis as an over-approximation of the main analysis, because an under-approximating pre-analysis would be too optimistic in context selection and the resulting selective main analysis is hardly cost-effective.

3. The abstract semantics of commands in the pre-analysis should be sound with respect to that of the main analysis:

$$\forall s \in \mathbb{S}, s^\sharp \in \mathbb{S}^\sharp.\ s \in \gamma(s^\sharp) \implies [\![cmd]\!](s) \in \gamma([\![cmd]\!]^\sharp(s^\sharp)).$$

4. The join operation of the pre-analysis's abstract domain over-approximates the widening operation of the main analysis: for all $X, Y \in \mathbb{D}$ and $X^\sharp, Y^\sharp \in \mathbb{D}^\sharp$,

$$(X \in \gamma(X^\sharp) \wedge Y \in \gamma(Y^\sharp)) \implies X \bigtriangledown Y \in \gamma(X^\sharp \sqcup Y^\sharp).$$

The purpose of our condition is that the impact pre-analysis over-approximates the fully context-sensitive main analysis:

Lemma 2 *Let $M \in \mathbb{D}$ be the main analysis result, i.e., a solution of* (3.5) *under full context-sensitivity ($K = K_\infty$). Let $P \in \mathbb{D}^\sharp$ be the pre-analysis result, i.e., the least solution of* (3.6)*. Then, $\forall c \in \mathbb{C}, \kappa \in \mathbb{C}_c^*.\ M(c, \kappa) \in \gamma(P(c, \kappa))$.*

Efficiency condition The next condition is for the efficiency of our pre-analysis. It consists of two requirements, and ensures that the pre-analysis can be computed using efficient algorithms:

1. The abstract states are $\bot$ or functions from program variables to abstract values: $\mathbb{S}^\sharp = \{\bot\} \cup (\mathsf{Var} \to \mathbb{V})$, where $\mathbb{V}$ is a finite complete lattice $(\mathbb{V}, \sqsubseteq_v, \bot_v, \top_v, \sqcup_v, \sqcap_v)$. An initial abstract state is $s_I^\sharp = \lambda x. \top_v$.

2. The abstract semantics of primitive commands has a simple form involving only join operation and constant abstract value, which is defined as follows:

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

where $[\![e]\!]^\sharp$ has the following form: for every $s \neq \bot$,

$$[\![e]\!]^\sharp(s) = s(x_1) \sqcup \ldots \sqcup s(x_n) \sqcup v$$

for some variables $x_1, \ldots, x_n$ and an abstract value $v \in \mathbb{V}$, all of which are fixed for the given e. We denote these variables and the value by

$$\mathsf{var}(e) = \{x_1, \ldots, x_n\}, \qquad \mathsf{const}(e) = v.$$

Example 3 (Impact Pre-Analysis for the Interval Analysis) *We design a pre-analysis for our interval analysis in Example 2, which satisfies our soundness and efficiency conditions. The pre-analysis aims at predicting which variables get associated with non-negative intervals when the program is analyzed by an interval analysis with full context-sensitivity K_∞.*

1. *Let $\mathbb{V} = \{\bot_v, \bigstar, \top_v\}$ be a lattice such that $\bot_v \sqsubseteq_v \bigstar \sqsubseteq_v \top_v$. Define the function $\gamma_v : \{\bot_v, \bigstar, \top_v\} \to \wp(\mathbb{I})$ as follows:*

   $$\gamma_v(\top_v) = \mathbb{I}, \quad \gamma_v(\bigstar) = \{[a, b] \in \mathbb{I} \mid 0 \leq a\}, \quad \gamma_v(\bot_v) = \emptyset$$

   This function determines the meaning of each element in $\mathbb{V}$ in terms of a collection of intervals. The only non-trivial case is $\bigstar$, which denotes all non-negative intervals according to this function. We include such a case because non-negative intervals, not negative ones, prove buffer-overrun properties.

2. *The domain of abstract states is defined as $\mathbb{S}^\sharp = \{\bot\} \cup (\mathsf{Var} \to \mathbb{V})$. The meaning of abstract states in $\mathbb{S}^\sharp$ is given by γ such that $\gamma(\bot) = \{\bot\}$ and, for $s^\sharp \neq \bot$,*

   $$\gamma(s^\sharp) = \{s \in \mathbb{S} \mid s = \bot \ \vee \ \forall x \in \mathsf{Var}.\ s(x) \in \gamma_v(s^\sharp(x))\}.$$

3. *Initial abstract state: $s_I^\sharp = \top = \lambda x.\top_v$.*

4. *Abstract evaluation $[\![e]\!]^\sharp$ of expression e: for every $s \neq \bot$,*

   $$\begin{array}{ll} [\![n]\!](s) = \mathrm{ite}(n \geq 0, \bigstar, \top_v), & [\![e_1 + e_2]\!](s) = [\![e_1]\!](s) \sqcup_v [\![e_2]\!](s) \\ [\![x]\!](s) = s(x), & [\![e_1 - e_2]\!](s) = \top_v \end{array}$$

The analysis approximately tracks numbers, but distinguishes the non-negative cases from general ones: non-negative numbers get abstracted to $\bigstar$ *by the analysis, but negative numbers are represented by* $\top_v$*. Observe that the* $+$ *operator is interpreted as the least upper bound* $\sqcup_v$*, so that* $e_1 + e_2$ *evaluates to* $\bigstar$ *only when both* e_1 *and* e_2 *evaluates to* $\bigstar$*. This implements the intuitive fact that the addition of two non-negative intervals gives another non-negative interval. For expressions involving subtractions, the analysis simply produces* $\top_v$*.*

Running the pre-analysis via reachability-based algorithm The class of our pre-analyses enjoys efficient algorithms (e.g., [10, 43]) for computing the least solution X that satisfies (3.6), even though it is fully context-sensitive. For our purpose, we provide a variant of the graph reachability-based algorithm in [43]. Our algorithm is specialized for our pre-analysis and is more efficient than the algorithm in [43]. Next, we go through each step of our algorithm while introducing concepts necessary to understand it. In the rest of this section, we interchangeably write K for K_∞.

First, our algorithm constructs the value-flow graph of the given program, which is a finite graph $(\Theta, \hookrightarrow)$ defined as follows:

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

The node set consists of pairs of program nodes and variables, and $(\hookrightarrow)$ is the edge relation between the nodes.

Definition 2 ($\hookrightarrow$) *The value-flow relation* $(\hookrightarrow) \subseteq (\mathbb{C} \times \mathsf{Var}) \times (\mathbb{C} \times \mathsf{Var})$ *links the vertices in* Θ *based on how values of variables flow to other variables in each primitive command:*

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

We can extend the $\hookrightarrow$ to its context-enriched version $\hookrightarrow_K$:

Definition 3 ($\hookrightarrow_K$) *The context-enriched value-flow relation* $(\hookrightarrow_K) \subseteq (\mathbb{C}_K \times \mathsf{Var}) \times (\mathbb{C}_K \times \mathsf{Var})$ *links the vertices in* $\mathbb{C}_K \times \mathsf{Var}$ *according to the specification below:*

$$
((c, \kappa), x) \hookrightarrow_K ((c', \kappa'), x') \quad \textit{iff}
$$
$$
\begin{cases}
(c, \kappa) \to_K (c', \kappa') \ \wedge \ x = x' & (\mathsf{cmd}(c') = \textit{skip}) \\
(c, \kappa) \to_K (c', \kappa') \ \wedge \ x = x' & (y \neq x') \\
(c, \kappa) \to_K (c', \kappa') \ \wedge \ x \in \mathsf{var}(e) & (y = x')
\end{cases}
$$
$$
\textit{(where } \mathsf{cmd}(c') \textit{ in the last two cases is } y := e\textit{)}
$$

Second, the algorithm computes the interprocedurally-valid reachability relation $(\hookrightarrow_K^\dagger) \subseteq \Theta \times \Theta$:

Definition 4 ($\hookrightarrow_K^\dagger$) *The reachability relation* $(\hookrightarrow_K^\dagger) \subseteq \Theta \times \Theta$ *connects two vertices when one node can reach the other via an interprocedurally-valid path:*

$$
\begin{aligned}
&(c, x) \hookrightarrow_K^\dagger (c', x') \quad \textit{iff} \\
&\exists \kappa, \kappa'. \, (\iota, \epsilon) \to_K^* (c, \kappa) \ \wedge \ ((c, \kappa), x) \hookrightarrow_K^* ((c', \kappa'), x').
\end{aligned}
$$

We use the tabulation algorithm in [43] for computing $(\hookrightarrow_K^\dagger)$. While computing $(\hookrightarrow_K^\dagger)$, the algorithm also collects the set C of reachable nodes:

$$
C = \{c \mid \exists \kappa. \, (\iota, \epsilon) \to_K^* (c, \kappa)\}. \tag{3.7}
$$

Third, our algorithm computes a set Θ_v of generators for each abstract value v in $\mathbb{V}$. Generators for v are vertices in Θ whose commands join v in their abstract semantics:

$$\begin{aligned}\Theta_v = \{(c,x) \mid \mathsf{cmd}(c) = x := e \wedge \mathsf{const}(e) = v\} \\ \cup\, (\mathit{if}\ (v = \top_v)\ \mathit{then}\ \{(\iota, x) \mid x \in \mathsf{Var}\}\ \mathit{else}\ \{\})\end{aligned}$$

Finally, using $(\hookrightarrow^{\dagger}_{K})$ and Θ_v, the algorithm constructs PA_K:

Definition 5 (PA_K) *$\mathsf{PA}_K \in \mathbb{C} \to \mathbb{S}^\sharp$ is defined as follows:*

$$\begin{aligned}\mathsf{PA}_K(c) = &\ \mathit{if}\ (c \notin C)\ \mathit{then}\ \bot \\ &\ \mathit{else}\ \lambda x. \bigsqcup \{v \in \mathbb{V} \mid \exists (c_0, x_0) \in \Theta_v.(c_0, x_0) \hookrightarrow^{\dagger}_{K} (c, x)\}.\end{aligned}$$

Then, PA_K is the solution of our pre-analysis:

Lemma 3 *Let X be the least solution satisfying* (3.6). *Then,* $\mathsf{PA}_K(c) = \bigsqcup_{\kappa \in \mathbb{C}^*} X(c, \kappa)$.

Our reachability-based algorithm is $|\mathbb{V}|^3$-times faster in the worst case than the RHS algorithm [43]. The algorithm in [43] works on a graph with the following set of vertices:

$$\Theta' = \{(c, s) \mid c \in \mathbb{C} \wedge s \neq \bot \wedge (\exists x. \forall y. y \neq x \implies s(y) = \bot_v)\}$$

Note that the set Θ' is $|\mathbb{V}|$-times larger than set Θ used in our algorithm and the worst-case time complexity is cubic on the size of the underlying graph [43].

3.5.2 Use of the Pre-Analysis Results

Using the pre-analysis results, we select queries that are likely to benefit from the increased context-sensitivity of the main analysis. Also, we collect calling contexts that are worth being distinguished during the main analysis. The collected contexts are used to construct a context selector K (Definition 10), which instructs how much context-sensitivity the main analysis should use for each procedure call. This main analysis with K is guaranteed to benefit from the increased context-sensitivity (Proposition 1).

Query selection We first select queries that can benefit from increased context-sensitivity. Among given queries $\mathcal{Q} \subseteq \mathbb{C} \times \mathbb{S} \times \mathsf{Var}$ about the given program, we select the following ones:

$$\begin{aligned}\mathcal{Q}^\sharp = \{ &(c, x) \in (\mathbb{C} \times \mathsf{Var}) \mid \exists s \in \mathbb{S}. \\ &(c, s, x) \in \mathcal{Q} \ \wedge \ \forall s' \in \gamma(\mathsf{PA}_{K_\infty}(c)).\ s \sqcup s' \neq \top\}\end{aligned} \tag{3.8}$$

where $\mathsf{PA}_{K_\infty} : \mathbb{C} \to \mathbb{S}^\sharp$ is the pre-analysis result. The first conjunct says that $(c, x) \in \mathcal{Q}^\sharp$ comes from some query $(c, s, x) \in \mathcal{Q}$, and the second conjunct expresses that according to the pre-analysis result, the main analysis does not lose too much information regarding this query. For instance, consider the case of interval analysis. In this case, we are usually interested in checking an assertion like $1 \leq x$ at c, which corresponds to a query (c, s, x) with the abstract state $s = (\lambda z.\ \text{if}\ (x = z)\ \text{then}\ [1, \infty]\ \text{else}\ \top)$. Then, the second conjunct in (3.8) becomes equivalent to $\mathsf{PA}_{K_\infty}(c)(x) \sqsubseteq \bigstar$. That is, we select the query only when the pre-analysis estimates that the variable x will have at least a non-negative interval in the main analysis. In the rest of this section, we assume for brevity that there is only one selected query $(c_q, x_q) \in \mathcal{Q}^\sharp$ in the program.

Building a context selector Next, we construct a context selector $\mathsf{K} : \mathbb{F} \to \wp(\mathbb{C}_c^*)$. K is to answer which calling contexts the main analysis should distinguish in order to achieve most of the benefits of context sensitivity on the given query (c_q, x_q). Our construction considers the following proxy of this goal: which contexts should *the pre-analysis* distinguish to achieve the same precision on the selected query (c_q, x_q) as in the case of the full context-sensitivity? In this subsection, we will define a context selector K (Definition 10) that answers this question (Proposition 1).

We construct K in two steps. Before giving our construction, we remind the reader that the impact pre-analysis works on the value-flow graph $(\Theta, \hookrightarrow)$ of the program and computes the reachability relation $(\hookrightarrow^\dagger_{K_\infty}) \subseteq \Theta \times \Theta$ over the interprocedurally-valid paths.

The first step is to build a program slice that includes all the dependencies of the query (c_q, x_q). A query (c_q, x_q) depends on a vertex (c, x) in the value-flow graph if there exists an interprocedurally-valid path between (c, x) and (c_q, x_q) on the graph (i.e., $(c, x) \hookrightarrow^{\dagger}_{K_\infty} (c_q, x_q)$). Tracing the dependency backwards from the query eventually hits vertices with no predecessors. We call such vertices *sources* and denote their set by Φ:

Definition 6 (Φ) *Sources Φ are vertices in Θ where dependencies begin:*

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

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

Definition 7 ($\Phi_{(c_q, x_q)}$) *Sources on which the query (c_q, x_q) depends:*

$$\Phi_{(c_q, x_q)} = \{(c_0, x_0) \in \Phi \mid (c_0, x_0) \hookrightarrow^{\dagger}_{K_\infty} (c_q, x_q)\}.$$

Example 4 *Consider the control flow graph in Figure 3.2. Node 6 denotes the query point, i.e., $(c_q, x_q) = (6, \mathtt{z})$. The gray nodes represent the sources on which the query depends, i.e., $\Phi_{(6,\mathtt{z})} = \{(1, \mathtt{x}), (7, \mathtt{y})\}$.*

For a source $(c_0, x_0) \in \Phi_{(c_q, x_q)}$ and an initial context κ_0 such that $(\iota, \epsilon) \rightarrow^{*}_{K_\infty} (c_0, \kappa_0)$, the following interprocedurally-valid path

$$((c_0, \kappa_0), x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q, \kappa_q), x_q) \tag{3.9}$$

represents a dependency path for the query (c_q, x_q). We denote the set of all dependency paths for the query by $\mathsf{Paths}_{(c_q, x_q)}$:

Definition 8 ($\mathsf{Paths}_{(c_q, x_q)}$) *The set of all dependency paths for the query (c_q, x_q) is defined as follows:*

$$\begin{aligned}\mathsf{Paths}_{(c_q, x_q)} = \{&((c_0, \kappa_0), x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q, \kappa_q), x_q) \\ &\mid (c_0, x_0) \in \Phi_{(c_q, x_q)} \ \wedge\ (\iota, \epsilon) \rightarrow^{*}_{K_\infty} (c_0, \kappa_0)\}.\end{aligned}$$

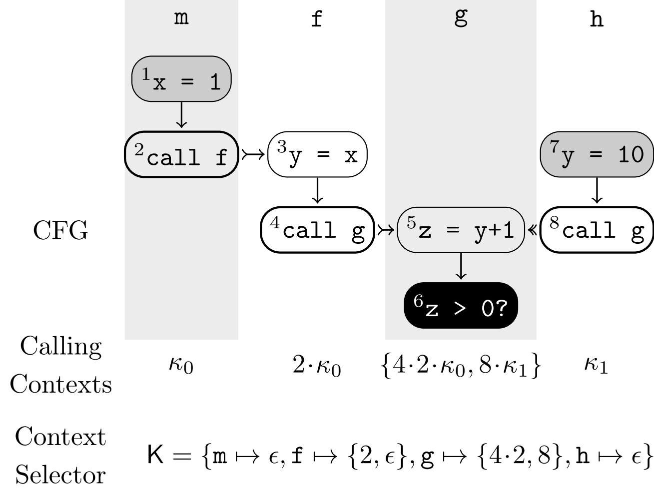


Figure 3.2: Example context selector. Gray and black nodes in CFG are source and query points, respectively.

$\mathsf{Paths}_{(c_q,x_q)}$ *is the program slice we intend to construct in this step.*

Example 5 *In Figure 3.2, suppose that κ_0 and κ_1 are the initial contexts of procedures* `m` *and* `h`*, respectively. For source $(1, \mathtt{x})$, we find the following dependency path to the query $(6, \mathtt{z})$:*

$$p_1 = ((1, \kappa_0), \mathtt{x}) \hookrightarrow_{K_\infty} ((2, \kappa_0), \mathtt{x}) \hookrightarrow_{K_\infty} ((3, 2 \cdot \kappa_0), \mathtt{y}) \hookrightarrow_{K_\infty} ((4, 2 \cdot \kappa_0), \mathtt{y}) \hookrightarrow_{K_\infty} ((5, 4 \cdot 2 \cdot \kappa_0), \mathtt{z}) \hookrightarrow_{K_\infty} ((6, 4 \cdot 2 \cdot \kappa_0), \mathtt{z})$$

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

$$p_2 = ((7, \kappa_1), \mathtt{y}) \hookrightarrow_{K_\infty} ((8, \kappa_1), \mathtt{y}) \hookrightarrow_{K_\infty} ((5, 8 \cdot \kappa_1), \mathtt{z}) \hookrightarrow_{K_\infty} ((6, 8 \cdot \kappa_1), \mathtt{z}).$$

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

The next step is to compute calling contexts that should be treated precisely. Consider a dependency path from $\mathsf{Paths}_{(c_q,x_q)}$:

$$((c_0, \kappa_0), x_0) \hookrightarrow_{K_\infty} \cdots \hookrightarrow_{K_\infty} ((c_q, \kappa_q), x_q) \tag{3.10}$$

where $\kappa_0, \kappa_1, \ldots, \kappa_q$ are the calling contexts appeared in the (fully context-sensitive) pre-analysis. Instead, we are interested in partial contexts that represent the "difference" between κ_i and κ_0. Intuitively, if κ_0 is a suffix of κ_i, i.e., $\kappa_i = \kappa'_i \cdot \kappa_0$, the partial context for κ_i is defined as κ'_i. Formally, we define the partial calling contexts of κ_i as $\kappa_i \ominus \kappa_0 = \kappa_i - \mathsf{suffix}(\kappa_i, \kappa_0)$ where $\mathsf{suffix}(\kappa_1, \kappa_2)$ is the longest common suffix of κ_1 and κ_2. For example, when κ_i is a suffix of κ_0, we use ϵ as the partial context for κ_i: if $\kappa_0 = c_2 \cdot c_1$ and $\kappa_i = c_1$, then $\kappa_i \ominus \kappa_0 = \epsilon$. Suppose that κ_i and κ_0 are not a suffix of each other, for instance $\kappa_0 = c_2 \cdot c_1$ and $\kappa_i = c_3 \cdot c_1$. In this case, $\kappa_i \ominus \kappa_0 = c_3$.

Assumption 1 *In general, the above definition of partial contexts requires that the input program should be well-formed with respect to the query: for a path* (3.10) *from a source to the query, every call site, $c_i \in \mathbb{C}_c$, in that path should not be included in the initial context κ_0. We require this condition because our selective context-sensitive analysis aims at distinguishing only the calls after passing the sources of dependency and analyzing context-insensitively those encountered before reaching those sources, which do not contribute to the query. This well-formedness assumption is not a strong restriction and its violation nearly never happens in practice. We did not observe any violation of the assumption in our benchmark programs (Section 5.5). If the program is not well-formed to a query, then we simply ignore it.*

Let us explain the condition with an example. Suppose that $\kappa_0 = c_3 \cdot c_2 \cdot c_1$ is the initial context at c_0 and $\kappa_i = c_1$ is the context at c_i. Suppose further that c_i is a call node. Then, our condition requires that c_i should not be one of call site c_1, c_2, and c_3. Formally, the condition is defined as follows:

We say the given program is well-formed with respect to the query (c_q, x_q) iff for every $(c_0, x_0) \in \Phi_{(c_q, x_q)}$ and its valid value-flow path

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

for all $0 \le i \le n$ *such that* $c_i \in \mathbb{C}_c$*,* c_i *is not included in the initial calling context* κ_0*; i.e.,*

$$c_i \notin \kappa_0 \tag{3.11}$$

where we write $c \in \kappa$ *when there exists some* κ' *such that* $c \cdot \kappa'$ *is a suffix of* κ*.*

In summary, for the path in (3.10), collecting contexts

$$\{\kappa_0 \ominus \kappa_0, \ldots, \kappa_q \ominus \kappa_0\}$$

give all the necessary partial calling contexts, where each $\kappa_i \ominus \kappa_0$ belongs to the calling contexts of procedure $\mathsf{fid}(c_i)$. Thus, we define the context selector for the dependency path (3.10) as follows:

Definition 9 (K_p, Context Selector for Path p) *Let* p *be a dependency path from a source* (c_0, x_0) *to query* (c_q, x_q)*:*

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

where κ_0 *is an initial context at* c_0 *such that* $(\iota, \epsilon) \rightarrow^*_{K_\infty} (c_0, x_0)$*. The context selector* K_p *for the path is defined as,*

$$\mathsf{K}_p = \lambda f.\ \{\kappa_i \ominus \kappa_0 \mid \mathsf{fid}(c_i) = f \ \wedge\ ((c_i, \kappa_i), _) \in p\}.$$

Example 6 *From the path* p_1 *in Example 5, the collection of* κ_i *is* $\{\kappa_0, 2{\cdot}\kappa_0, 4{\cdot}2{\cdot}\kappa_0\}$ *(see Figure 3.2). Hence, the collection of* $\kappa_i \ominus \kappa_0$ *is* $\{\epsilon, 2, 4 \cdot 2\}$*, where* ϵ *belongs to procedure* `m`*,* 2 *to* `f`*, and* $4 \cdot 2$ *to* `g`*. Similar for path* p_2*. Thus,* K_{p_1} *and* K_{p_2} *are:*

$$\mathsf{K}_{p_1} = \begin{bmatrix} \mathtt{m} & \mapsto & \{\epsilon\} \\ \mathtt{f} & \mapsto & \{2\} \\ \mathtt{g} & \mapsto & \{4 \cdot 2\} \end{bmatrix} \quad \mathsf{K}_{p_2} = \begin{bmatrix} \mathtt{h} & \mapsto & \{\epsilon\} \\ \mathtt{g} & \mapsto & \{8\} \end{bmatrix}$$

Then, the final context selector K is the union of K_p's:

Definition 10 (K, Context Selector) *Let (c_q, x_q) be a query. The context selector $\mathsf{K} \in \mathbb{F} \to \wp(\mathbb{C}_c^*)$ for our selective analysis is:*

$$\mathsf{K}(f) \;=\; \mathcal{E}(f) \cup \bigcup\{\mathsf{K}_p(f) \mid p \in \mathsf{Paths}_{(c_q,x_q)}\} \tag{3.12}$$

where $\mathcal{E}(f) = \{\epsilon\}$ if $f \neq \mathsf{fid}(c_q)$; and otherwise, $\mathcal{E}(f) = \emptyset$.

Running selective context-sensitive main analysis Finally, we run the main analysis with selective context-sensitivity K defined by the result of the impact pre-analysis. The following proposition states that the pre-analysis-guided context-sensitivity (K) manages to pay off at the selective main analysis, although the pre-analysis is fully context-sensitive and the main analysis is not.

Proposition 1 (Impact Realization) *Let $\mathsf{PA}_{K_\infty} \in \mathbb{C} \to \mathbb{S}^\sharp$ be the result of the impact pre-analysis (Definition 5). Let $q \in \mathcal{Q}^\sharp$ be a selected query (3.8). Let K be the context selector for q (Definition 10) defined using the pre-analysis result PA_{K_∞}. Let $\mathsf{MA}_\mathsf{K} \in \mathbb{C}_K \to \mathbb{S}$ be the main analysis result with the context selector K. Then, the selective main analysis is at least as precise as the fully context-sensitive pre-analysis for the selected query q:*

$$\mathsf{MA}_\mathsf{K} \sqsubseteq_q \mathsf{PA}_{K_\infty}$$

where $\mathsf{MA}_\mathsf{K} \sqsubseteq_q \mathsf{PA}_{K_\infty}$ iff $(q \overset{let}{=} (c, x))$

$$\forall \kappa \in \mathsf{K}(\mathsf{fid}(c)).\ \mathsf{MA}_\mathsf{K}(\kappa, c) \in \gamma(\top[x \mapsto \mathsf{PA}_{K_\infty}(c)(x)]).$$

This impact realization holds thanks to two key properties. First, our selective context-sensitivity K (Definition 10) distinguishes all the calling contexts that matter for the queries selected by the pre-analysis. Second, the main analysis designed in Section 3.4 isolates these distinguished contexts from other undistinguished contexts (ϵ), ensuring that spurious flows caused by merging contexts never adversely affect the precision of the selected query.

3.6 Application to Selective Relational Analysis

A general principle behind our method is that we can selectively improve the precision of the analysis by using an impact pre-analysis that estimates the main static analysis of the maximal precision. In this section, we use the same principle to develop a selective relational analysis with the octagon domain [30].

Overview Consider the following code snippet:

```
1  int a = b;
2  int c = input();          // User input
3  for (i = 0; i < b; i++) {
4    assert (i < a);         // Query 1
5    assert (i < c);         // Query 2
6  }
```

The first query at line 4 always holds but the second one at line 5 is not necessarily true.

A fully relational octagon analysis, which tracks constraints of the form $\pm x \pm y \leq c$ (where $c \in \mathbb{Z} \cup \{\infty\}$) between *all* variables x and y, can prove the first query. The analysis infers constraints $\mathtt{b} - \mathtt{a} \leq 0$ at line 1 and $\mathtt{i} - \mathtt{b} \leq -1$ at line 3. Then, combining the two via a closure operation [30], the analysis concludes that constraint $\mathtt{i} - \mathtt{a} \leq -1$ holds at line 4. More specifically, the fully relational octagon analysis computes the table (i.e., difference bound matrix [30]) on the left side of the following:

	a	b	c	i
a	0	0	∞	-1
b	0	0	∞	-1
c	∞	∞	0	∞
i	∞	∞	∞	0

	a	b	c	i
a	$\bigstar$	$\bigstar$	$\top$	$\bigstar$
b	$\bigstar$	$\bigstar$	$\top$	$\bigstar$
c	$\top$	$\top$	$\bigstar$	$\top$
i	$\top$	$\top$	$\top$	$\bigstar$

(3.13)

where the bound c in constraint $x - y \leq c$ is stored at row y and column x in the table.[2] Note that the (a,i) entry of the table stores -1, which means that the analysis proves $\texttt{i} - \texttt{a} \leq -1$ at line 4.

However, this fully relational analysis tracks unnecessary relationships between variables, which are either irrelevant to the query or not beneficial to the analysis precision. For instance, it is sufficient to keep only the contraints between a, b, and i to prove the first query, but the analysis unnecessarily maintains other relationships such as one between a and c. Besides, tracking a relationship between, for example, i and c does not change the end result of the analysis because the second query is impossible to prove.

Our selective octagon analysis tracks octagon constraints only when doing so is likely to improve the precision that matters for resolving given queries. To achieve this goal, we use an impact pre-analysis that aims at estimating the behavior of the octagon analysis under its fully relational setting. More specifically, like the fully relational octagon analysis, the pre-analysis tracks constraints of the form $\pm x \pm y \leq a$ for *all* variables x and y but approximately tracks the bound; we use one of two abstract values $\bigstar$ and $\top$ as bound a, rather than all integers and ∞. Here $x + y \leq \top$ represents all octagon constraints of the form $x + y \leq c$ including the case that $c = \infty$, whereas $x + y \sqsubseteq \bigstar$ means octagon constraints $x + y \leq c$ with integer constant c. This simple abstract domain is chosen because constant bound, not ∞, proves buffer-overrun properties. For instance, in our example program, the pre-analysis result at line 4 is the table on the righthand side in (3.13).

Next, using the pre-analysis results, we select variables whose relationships help improve the precision regarding given queries. We first identify queries (in our example, the first query) whose values are evaluated to $\bigstar$ using the pre-analysis results. Then, for each of selected queries, we do a dependency analysis to find out the variables whose relationships should be tracked together for the main analysis to answer query. For instance, consider that the constraint regarding the first query is $\texttt{i} - \texttt{a} \sqsubseteq \bigstar$. Our dependency analysis figures out that the constraint was derived

[2] For simplicity, we consider only constraints of the form $x - y \leq c$. In fact, the octagon analysis tracks constraints of both forms $x - y \leq c$ and $x + y \leq c$ and maintains a matrix of size $(2 \times |\mathsf{Var}|)^2$.

in the pre-analysis by combining two constraints $\mathtt{i} - \mathtt{b} \sqsubseteq \bigstar$ and $\mathtt{b} - \mathtt{a} \sqsubseteq \bigstar$ in its closure operation. Therefore, the dependency analysis concludes that the main analysis should be able to derive three relationships $\mathtt{i} - \mathtt{a} \sqsubseteq \bigstar$, $\mathtt{i} - \mathtt{b} \sqsubseteq \bigstar$, and $\mathtt{b} - \mathtt{a} \sqsubseteq \bigstar$ to prove the first query. Based on this conclusion, our selective octagon analysis decides to track the relationships between variables a, b, and i.

In the rest of this section, we formalize the key aspects of our selective octagon analysis.

Selective octagon analysis We first specify selective octagon analyses for the following simple commands:

$$cmd \;\to\; x := y + k \mid x :=?$$

where $k \in \mathbb{Z}$ is a positive integer and ? models arbitrary integers. We use Miné's definitions [30] of the octagon domain $\mathbb{O}$ and abstract semantics $[\![cmd]\!] : \mathbb{O} \to \mathbb{O}$ of primitive commands; we consider the positive form x and negative form $\bar{x}$ for each variable x and represent an octagon domain element $o \in \mathbb{O}$ by a $2|\mathsf{Var}| \times 2|\mathsf{Var}|$ matrix where each entry $o_{xy} \in \mathbb{Z} \cup \{+\infty\}$ stores the upper bound of $y - x$. The definition of $[\![cmd]\!]$ for our commands can be found at [30].

With $\mathbb{O}$ and $[\![cmd]\!]$, we define the domain of *packed octagons* that assign an octagon to a subset of variables, which we call *pack*. An octagon of a pack expresses only the constraints of the variables in that pack. We call $\Pi \subseteq \wp(\mathsf{Var})$ of sets of variables *packing configuration*, such that $\bigcup \Pi = \mathsf{Var}$. The packed octagon domain $\mathbb{O}_\Pi(\Pi)$ parameterized by packing configuration Π is then defined as $\mathbb{O}_\Pi(\Pi) = \Pi \to$

$\mathbb{O}$. We extend the abstract semantics $[\![cmd]\!] : \mathbb{O} \to \mathbb{O}$ of command cmd to $[\![cmd]\!]^{\Pi}$: $\mathbb{O}_{\Pi}(\Pi) \to \mathbb{O}_{\Pi}(\Pi)$ as follows:

$$
\begin{aligned}
&[\![c]\!]^{\Pi}(po) = \lambda\pi \in \Pi.\\
&\begin{cases}
[\![x := y + k]\!](po(\pi)) & (c = x := y + k \ \wedge \ x \in \pi \ \wedge \ y \in \pi)\\
[\![x :=?]\!](po(\pi)) & (c = x := y + k \ \wedge \ x \in \pi \ \wedge \ y \notin \pi)\\
[\![x :=?]\!](po(\pi)) & (c = x :=? \ \wedge \ x \in \pi)\\
po(\pi) & otherwise
\end{cases}
\end{aligned}
$$

The extended abstract semantics is essentially the same except it forgets all the relationships of the assignee x (the second case) when the pack is missing one variable involved in the octagonal constraint. The abstract semantics of program in $\mathbb{D} = \mathbb{C} \to \mathbb{O}_{\Pi}(\Pi)$ is defined as the least fixpoint of abstract transfer function $F^{\Pi} : \mathbb{D} \to \mathbb{D}$, which is defined as usual.

The selectivity of the analysis is governed by the configuration Π. For instance, with $\Pi = \{\{x\} \mid x \in \mathsf{Var}\}$, the analysis degenerates to a non-relational analysis. With $\Pi = \{\mathsf{Var}\}$, the analysis becomes a fully relational analysis. Our goal is to find a cost-effective Π by using an impact pre-analysis.

Impact pre-analysis Second, we formally define the impact pre-analysis. The meaning of our abstract values ($\mathbb{V} = \{\bigstar, \top\}$) is described by $\gamma_{\mathbb{V}}$ such that $\gamma_{\mathbb{V}}(\bigstar) = \mathbb{Z}$ and $\gamma_{\mathbb{V}}(\top) = \mathbb{Z} \cup \{+\infty\}$. The abstract state $\mathbb{O}^{\sharp} = \{\bot^{\sharp}\} \cup \mathbb{V}^{2|\mathsf{Var}| \times 2|\mathsf{Var}|}$ of our pre-analysis is the set of matrices whose entries are in $\mathbb{V}$. An abstract state $o^{\sharp} \in \mathbb{O}^{\sharp}$ denotes a set of octagons: we define $\gamma : \mathbb{O}^{\sharp} \to \wp(\mathbb{O})$ as follows:

$$\gamma(o^{\sharp}) = \{o \in \mathbb{O} \mid \forall i, j.\ o_{ij} \in \gamma_{\mathbb{V}}(o^{\sharp}_{ij})\}$$

The abstract semantics $[\![cmd]\!]^\sharp : \mathbb{O}^\sharp \to \mathbb{O}^\sharp$ of each primitive command cmd of the pre-analysis is defined as an over-approximation of the abstract semantics of the main analyses: e.g.,

$$\left([\![x :=?]\!]^\sharp(o^\sharp)\right)_{ij} = \begin{cases} \bigstar & (i = j = x \text{ or } i = j = \bar{x}) \\ \top & (i \notin \{x, \bar{x}\} \text{ or } j \notin \{x, \bar{x}\}) \\ o^\sharp_{ij} & \textit{otherwise} \end{cases}$$

The abstract domain of the pre-analysis is $\mathbb{D}^\sharp = \mathbb{C} \to \mathbb{O}^\sharp$ and the pre-analysis result is defined as the least fixpoint of semantic function $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$, which is defined as usual.

Use of pre-analysis results From the pre-analysis results $(\mathsf{lfp} F^\sharp)$, we construct Π as follows. Assume that a set $\mathcal{Q} \subseteq \mathbb{C} \times \mathsf{Var} \times \mathsf{Var}$ of relational queries is given in the program. A query $(c, x, y) \in \mathcal{Q}$ represents a predicate $y - x < 0$ at program point c and we say that $o \in \mathbb{O}$ proves the query when $o_{xy} \leq -1$. We first select a set $\mathcal{Q}^\sharp$ of queries that are judged promising by the pre-analysis:

$$\mathcal{Q}^\sharp = \{(c, x, y) \in \mathcal{Q} \mid (\mathsf{lfp} F^\sharp)(c) \neq \bot^\sharp \wedge (\mathsf{lfp} F^\sharp)(c)_{xy} = \bigstar\}.$$

Next, for each selected query $(c, x, y) \in \mathcal{Q}^\sharp$, we compute the pack $\pi_{(c,x,y)} \subseteq \mathsf{Var}$ of necessary variables using dependency analysis, which is simultaneously done with the pre-analysis as follows: let $\mathbb{V}^\natural$ be $\mathbb{V} \times \wp(\mathsf{Var})$ and $\mathbb{O}^\natural$ be the set of $2|\mathsf{Var}| \times 2|\mathsf{Var}|$ matrices over $\mathbb{V}^\natural$. The idea is to over-approximate the involved variables for each octagon constraint in the second component of $\mathbb{V}^\natural$. The abstract semantics $[\![\cdot]\!]^\natural : \mathbb{O}^\natural \to \mathbb{O}^\natural$ is the same as $[\![\cdot]\!]^\sharp$ except that it also maintains the involved variables: e.g.,

$$\left([\![x :=?]\!]^\natural(o^\natural)\right)_{ij} = \begin{cases} \left(\bigstar, \{i, j\}\right) & (i = j = x \text{ or } i = j = \bar{x}) \\ (\top, \mathsf{Var}) & (i \notin \{x, \bar{x}\} \text{ or } j \notin \{x, \bar{x}\}) \\ {o^\natural}_{ij} & \textit{otherwise} \end{cases}$$

Let $F^\natural : (\mathbb{C} \to \mathbb{O}^\natural) \to (\mathbb{C} \to \mathbb{O}^\natural)$ be the abstract transfer function and $\mathsf{lfp}F^\natural$ be its least fixpoint. Then, the pack $\pi_{(c,x,y)}$ is defined as S such that $\big((\mathsf{lfp}F^\natural)(c)\big)_{xy} = \big(\bigstar, S\big)$. Finally, we extract the packing configuration Π using $\pi_{(c,x,y)}$ as follows:

$$\Pi = \{\pi_{(c,x,y)}\} \cup \{\{z\} \mid z \in \mathsf{Var} \setminus \pi_{(c,x,y)}\}. \tag{3.14}$$

Selective main octagon analysis We run the selective octagon analysis with the packing configuration in (3.14).

Proposition 2 (Impact Realization) *Let $\pi_{(c,x,y)}$ be the pack for query (c, x, y) defined by the result of our impact pre-analysis. Let Π be the packing configuration for $\pi_{(c,x,y)}$, which is defined in (3.14). Let F^Π be the transfer function of the selective octagon analysis with the Π. Then, $\big((\mathsf{lfp}F^\Pi)(c)(\pi_{(c,x,y)})\big)_{xy} \neq +\infty$.*

3.7 Experiments

Selective Context-Sensitive Analysis In experiments, we used Sparrow [54], a buffer-overrun analyzer that supports the full set of the C language. The baseline analyzer performs a flow-sensitive and context-insensitive analysis, and tracks both numeric and pointer values. For numeric values, it uses the interval domain by default (alternatively, it can use the octagon domain). In addition to the interval domain, the analysis uses an allocation-site–based heap abstraction for dynamic memory allocation.

On top of the baseline analyzer, we have implemented our technique: we implemented the impact pre-analysis in Example 3 and extended the baseline analysis to be selectively context-sensitive. In Section 3.5.2, we considered only one query; in implementation, the pre-analysis computes a single context selector K that specifies calling contexts for multiple queries. When analyzing a procedure under different calling contexts, we distinguish allocation sites for each context; that is, an allocation-site produces different abstract locations under different calling contexts.

We have run the analysis for 10 software packages from the GNU open-source projects. The analysis is global: the entire program is analyzed starting from the

`main` procedure. All experiments were done on a Linux 2.6 system running on a single core of Intel 3.07GHz box with 24GB of memory.

Table 3.1 presents the performance of our selective context-sensitive analysis and compares it with the context-insensitive analysis. We measured the analysis precision by the number of buffer accesses (#**alarm**) that cannot be proven safe by the analysis.

The results show that our method leads to a cost-effective improvement of the analysis precision. In total, the context-insensitive interval analysis points out 12,701 buffer accesses as potential buffer-overrun errors (there is a total of 83,776 buffer accesses in the 10 programs). Our technique reduces the number down to 9,598 (24.4% reduction). In doing so, our technique increases the total analysis time from 707.1s to 903.6s (27.8% increase).

We observed that passing numeric values through long call chains is not uncommon in the interval analysis of C programs. Our pre-analysis is able to prescribe such a long call sequence as context-sensitive targets. For instance, in `a2ps-4.14`, among 1682 call sequences prescribed by our pre-analysis, 488 call sequences were of length longer than or equal to 3.

According to our experience, the k-callstrings approach does not scale when it is used with the interval abstract domain for analyzing C programs. The 2- and 3-callstrings approaches did not stop after 30 minutes for programs over 10KLOC. Even the 1-callstrings approach was slow and did not scale over 40KLOC. For instance, the 3-callstrings approach succeeded to analyze `spell-1.0` in 11.9s (with 30 alarms reported), it did not stop for `bc-1.06`.

Selective Octagon Analysis We have implemented our selective method on top of the octagon-analysis version of our baseline analyzer. We compare the performance of our selective analysis with an existing octagon analysis based on the syntactic variable packing [30,36]. The syntactic packing approach relates variables together if they are involved in the same syntactic block [30]. We limited the maximum pack size by 10 in the syntactic packing strategy, since otherwise the analysis did not scale.

Table 3.2 shows our benchmark programs. Note that, although a relational analysis is a key to proving important numerical properties, it is useful only for specific target programs and queries [11, 30]. Thus, we first identified a set of benchmark programs and their buffer-overrun queries whose proofs require relational information, and compared the performance of the two analyses on these programs and queries. Column **#Q** shows the number of relational queries that we consider in our experiments. In the experiments, we manually in-lined the functions that are involved in the proofs of the target queries, so that our selective relational analysis and the syntactic packing approach are run under context-sensitivity.

The results show that our selective octagon analysis has a competitive precision-cost balance. Among 135 queries in total, our analysis is able to prove 132 (97.8%) queries in 3,632.7s. On the other hand, the octagon analysis with syntactic packing proved 44 (32.6%) queries in 33,840.3s; the syntactic packing heuristic often fails to prescribe variable relationships necessary to prove queries. Our analysis is even faster than the counterpart in most cases because it selectively turns on relational analysis.

One thing to note is that running our pre-analysis is feasible in practice even though it is fully relational. The bottlenecks of a fully relational octagon analysis are the memory costs for representing $2|\mathsf{Var}| \times 2|\mathsf{Var}|$ matrices and the expensive strong closure operation [30] whose time complexity is cubic in the number of variables. Thanks to the simplicity of the abstract domain ($\bigstar$ or $\top$), we can reduce the memory cost using a sparse representation for the matrices. For the closure operation, we use Dijkstra's algorithm and compute the shortest-path closure [30] instead of the strong closure. In our experiments, using the shortest-path closure made no difference in the pre-analysis precision.

3.8 Summary

We proposed a method of designing a selective "X-sensitive" analysis, where the selection is guided by an impact pre-analysis. We followed this approach, presented two program analyses that selectively apply precision-improving techniques, and

demonstrated their effectiveness with experiments in a realistic setting. The first was a selective context-sensitive analysis that receives guidance from an impact pre-analysis. Our experiments with realistic benchmarks showed that the method reduces the number of false alarms of a context-insensitive interval analysis by 24.4%, while increasing the analysis cost by 27.8%. The second example was a selective relational analysis with octagons using the same idea of impact pre-analysis, and our experiments showed that our selective octagon analysis proves 88 more queries than the existing one based on the syntactic variable packing and reduces the analysis cost by 81%. We believe that our approach can be used for developing other selective analyses as well, e.g., selective flow-sensitive analysis, selective loop-unrolling, etc.

Program	LOC	Proc	Context-Insensitive		Our Selective Context-Sensitive Analysis						Alarm	Overhead	
			#alarm	time	#alarm	pre	main	total	#selected call-sites	$\rightsquigarrow$	reduction	pre	main
spell-1.0	2,213	31	58	0.6	30	0.1	0.8	0.9	25 / 124 (20.2 %)	3	48.3%	16.7%	33.3%
bc-1.06	13,093	134	606	14.0	483	1.9	14.3	16.2	29 / 777 (3.7 %)	2	20.3%	13.6%	2.1%
tar-1.17	20,258	222	940	42.1	799	5.4	41.8	47.2	51 / 1213 (4.2 %)	3	15.0%	12.8%	−0.7%
less-382	23,822	382	654	123.0	562	3.3	163.1	166.4	51 / 1,522 (3.4 %)	4	14.1%	2.7%	32.6%
sed-4.0.8	26,807	294	1,325	107.5	1,238	7.4	110.2	117.6	25 / 868 (2.9 %)	3	6.6%	6.9%	2.5%
make-3.76	27,304	191	1,500	84.4	1,028	7.1	99.1	106.2	67 / 1,050 (6.4 %)	3	31.5%	8.4%	17.4%
grep-2.5	31,495	153	735	12.1	653	2.4	13.5	15.9	33 / 530 (6.2 %)	3	11.2%	19.8%	11.6%
wget-1.9	35,018	434	1,307	69.0	942	12.5	69.6	82.1	79 / 1,973 (4.0 %)	5	27.9%	18.1%	0.9%
a2ps-4.14	64,590	980	3,682	118.1	2,121	29.5	148.2	177.7	237 / 2,450 (9.7 %)	9	42.4%	25.0%	25.5%
bison-2.5	101,807	1,427	1,894	136.3	1,742	34.6	138.8	173.4	173 / 2,038 (8.5 %)	4	8.0%	25.4%	1.8%
Total	346,407	4,248	12,701	707.1	9,598	104.2	799.4	903.6	770 / 12,545 (6.1 %)		**24.4%**	**14.7%**	**13.1%**

Table 3.1: Performance comparison between context-insensitive analysis and our selective context-sensitive analysis. **LOC** reports lines of code before pre-processing. **Proc** shows the number of procedures in the programs. **#alarm** reports the number of buffer-overrun alarms raised by the analyses. **pre** reports the time spent for running the pre-analysis (including query selection and building context selector) and **main** reports the time spent by the main analysis of our approach. Each entry a/b (c%) in column **#selected call-sites** means that, among b call-sites in the program, a call-sites are selected for context-sensitivity by our pre-analysis and the selection ratio is c%. $\rightsquigarrow$ reports the maximum call-depth prescribed by the pre-analysis. **Overhead**: **pre** shows the pre-analysis overhead and **main** reports the cost increase in the main analysis due to increased context-sensitivity, compared to the context-insensitive analysis.

Program	**LOC**	**#Var**	**#Q**	**Syntactic Packing Approach**				**Our Selective Relational Analysis**						**Comparison**	
				proven	**time**	**mem**	**pack**	**proven**	**pre**	**main**	**total**	**mem**	**pack**	**Precision**	**Time**
calculator-1.0	298	197	10	2	0.3	63	18 (7.3)	10	0.1	0.1	0.2	52	3 (3.6)	+8	-33.3%
spell-1.0	2,213	531	16	1	4.8	109	119 (7.7)	16	1.7	0.7	2.4	63	6 (11.0)	+15	-50.0%
barcode-0.96	4,460	2,002	37	16	11.8	221	276 (8.1)	37	12.2	18.3	30.5	100	12 (25.0)	+21	158.5%
httptunnel-3.3	6,174	1,908	28	16	26.0	220	454 (7.0)	26	10.8	4.5	15.3	105	8 (5.8)	+10	-41.2%
bc-1.06	13,093	2,194	10	2	247.1	945	606 (7.8)	9	82.3	35.0	117.3	212	4 (4.0)	+7	-52.5%
tar-1.17	20,258	5,332	17	7	1,043.2	1,311	1,259 (7.5)	17	598.5	63.3	661.8	384	7 (3.9)	+10	-36.6%
less-382	23,822	4,482	13	0	3,031.5	1,439	1,017 (6.3)	13	2,253.2	596.2	2,849.4	955	8 (6.3)	+13	-6.0%
a2ps-4.14	64,590	16,531	11	0	29,479.3	2,304	2,608 (7.8)	11	2,223.5	518.2	2,741.7	909	6 (6.7)	+11	-90.7%
Total	135,008	33,177	142	44	33,840.3	6,611		139	5,182.3	1,236.3	6,418.6	2,780		+95	-81.0%

Table 3.2: Performance comparison between an octagon analysis with an existing syntactic packing strategy and our selective relational analysis. #**Var** denotes the number of variables (abstract locations) in the program. #**Q** denotes the number of buffer-overrun queries whose proofs require relational reasoning. **proven** reports the number of queries that are proven by each octagon analysis. **mem** reports the peak memory consumption in megabytes. Each X (Y) in column **pack** represents the number of non-singleton packs (X) and the average size (Y) of the packs used in each relational analysis. **Precision** and **Time** shows additionally proven queries and time consumption by our selective relational analysis compared to the syntactic packing approach.

Chapter 4

Selectively X-sensitive analysis by learning data generated by impact pre-analysis

4.1 Introduction

Relational program analyses track sophisticated relationships among program variables and enable the automatic verification of complex properties of programs [9, 30]. However, the computational costs of various operations of these analyses are high so that vanilla implementations of the analyses do not scale even to moderate-sized programs. For example, transfer functions of the Octagon analysis [30] have a cubic worst-case time complexity in the number of program variables, which makes it impossible to analyze large programs.

In this paper, we consider one of the most popular optimizations used by practical relational program analyses, called variable clustering [2, 30, 37, 57]. Given a program, an analyzer with this optimization forms multiple relatively-small subsets of variables, called variable clusters or clusters. Then, it limits the tracked information to the relationships among variables within each cluster, not across those clusters. So far strategies based on simple syntactic or semantic criteria have been used

for clustering variables for a given program, but they are not satisfactory. They are limited to a specific class of target programs [2, 57] or employ a pre-analysis that is cheaper than a full relational analysis but frequently takes order-of-magnitude more time than the non-relational analysis for medium-sized programs [37].

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

The most important aspect of our learning method is the automatic provision of labeled data. Although the method is essentially an instance of supervised learning, it does not require the common unpleasant involvement of humans in supervised learning, namely, labeling. Our method takes a codebase consisting of typical programs of small-to-medium size, and automatically generates labels for pairs of variables in those programs by using the impact pre-analysis from our previous work [37], which estimates the impact of tracking relationships among variables by Octagon on proving queries in given programs. Our method precisely labels a pair of program variables with $\oplus$ when the pre-analysis says that the pair should be tracked. Because this learning occurs offline, we can bear the cost of the pre-analysis, which is still significantly lower than the cost of the full Octagon analysis. Once labeled data are generated, our method runs an off-the-shelf classification algorithm, such as decision-tree inference [31], for inferring a classifier for those labeled data. This classifier is used to map vector representations of variable pairs

to $\oplus$ or $\ominus$. Conceptually, the inferred classifier is a further approximation of the pre-analysis, which gets found automatically from a given codebase.

The experimental results show that our method results in the learning of a cost-effective variable-clustering strategy. We implemented our learning method on top of a static buffer overflow detector for C programs and tested against open source benchmarks. In the experiments, our analysis with the learned variable-clustering strategy scales up to 100KLOC within the two times of the analysis cost of the Interval analysis. This corresponds to the 33x speed-up of the selective relational analysis based on the impact pre-analysis [37] (which was already significantly faster than the original Octagon analysis). The price of speed-up was mere 2% increase of false alarms.

We summarize the contributions of this paper below:

1. We propose a method for automatically learning an effective strategy for variable-clustering for the Octagon analysis from a given codebase. The method infers a function that decides, for a program P and a pair of variables (x, y) in P, whether tracking the relationship between x and y is important. The learned strategy uses this function to cluster variables in a given program.

2. We show how to automatically generate labeled data from a given codebase that are needed for learning. Our key idea is to generate such data using the impact pre-analysis for Octagon from [37]. This use of the pre-analysis means that our learning step is just the process of finding a further approximation of the pre-analysis, which avoids expensive computations of the pre-analysis but keeps its important estimations.

3. We experimentally show the effectiveness of our learning method using a realistic static analyzer for full C and open source benchmarks. Our variant of Octagon with the learned strategy is 33x faster than the selective relational analysis based on the impact pre-analysis [37] while increasing false alarms by only 2%.

4.2 Informal Explanation

4.2.1 Octagon Analysis with Variable Clustering

We start with informal explanation of our approach using the program in Figure 4.1. The program contains two queries about the relationships between `i` and variables `a`, `b` inside the loop. The first query `i` $<$ `a` is always true because the loop condition ensures `i` $<$ `b` and variables `a` and `b` have the same value throughout the loop. The second query `i` $<$ `c`, on the other hand, may become false because `c` is set to an unknown input at line 2.

The Octagon analysis [30] discovers program invariants strong enough to prove the first query in our example. At each program point it infers an invariant of the form

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

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

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

(4.1)

The ij-th entry m_{ij} of this matrix means an upper bound $e_j - e_i \leq m_{ij}$, where e_j and e_i are expressions associated with the j-th column and the i-th row of the matrix respectively and they are variables with or without the minus sign. The

```
1  int a = b;
2  int c = input();          // User input
3  for (i = 0; i < b; i++) {
4    assert (i < a);         // Query 1
5    assert (i < c);         // Query 2
6  }
```

Figure 4.1: Example program

matrix records -1 and ∞ as upper bounds for $\mathtt{i} - \mathtt{a}$ and $\mathtt{i} - \mathtt{c}$, respectively. Note that these bounds imply the first query, but not the second.

In practice the Octagon analysis is rarely used without further optimization, because it usually spends a large amount of computational resources for discovering unnecessary relationships between program variables, which do not contribute to proving given queries. In our example, the analysis tracks the relationship between `c` and `i`, although it does not help prove any of the queries.

A standard approach for addressing this inefficiency is to form subsets of variables, called variable clusters or clusters. According to a pre-defined clustering strategy, the analysis tracks the relationships between only those variables within the same cluster, not across clusters. In our example, this approach would form two clusters $\{\mathtt{a}, \mathtt{b}, \mathtt{i}\}$ and $\{\mathtt{c}\}$ and prevent the Octagon analysis from tracking the unnecessary relationships between `c` and the other variables. The success of the approach lies in finding a good strategy that is able to find effective clusters for a given program. This is possible as demonstrated in the several previous work [2, 37, 57], but it is highly nontrivial and often requires a large amount of trial and error of analysis designers.

Our goal is to develop a method for automatically learning a good variable-clustering strategy for a target class of programs. This automatic learning happens offline with a collection of typical sample programs from the target class, and the learned strategy is later applied to any programs in the class, most of which are not used during learning. We want the learned strategy to form relatively-small

variable clusters so as to lower the analysis cost and, at the same time, to put a pair of variables in the same cluster if tracking their relationship by Octagon is important for proving given queries. For instance, such a strategy would cluster variables of our example program into two groups $\{\texttt{a}, \texttt{b}, \texttt{i}\}$ and $\{\texttt{c}\}$, and make Octagon compute the following smaller matrix at the first query:

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

(4.2)

4.2.2 Automatic Learning of a Variable-Clustering Strategy

In this paper we will present a method for learning a variable-clustering strategy. Using a given codebase, it infers a function $\mathcal{F}$ that maps a tuple $(P, (x, y))$ of a program P and variables x, y in P to $\oplus$ and $\ominus$. The output $\oplus$ here means that tracking the relationship between x and y is likely to be important for proving queries. The inferred $\mathcal{F}$ guides our variant of the Octagon analysis. Given a program P, our analysis applies $\mathcal{F}$ to every pair of variables in P, and computes the finest partition of variables that puts every pair (x, y) with the $\oplus$ mark in the same group. Then, it analyzes the program P by tracking relationships between variables within each group in the partition, but not across groups.

Our method for learning takes a codebase that consists of typical programs in the intended application of the analysis. Then, it automatically synthesizes the above function $\mathcal{F}$ in two steps. First, it generates labeled data automatically from programs in the codebase by using the impact pre-analysis for Octagon from our previous work [37]. This is the most salient aspect of our approach; in a similar supervised-learning task, such labeled data are typically constructed manually, and avoiding this expensive manual labelling process is considered a big challenge for

supervised learning. Next, our approach converts labeled data to boolean vectors marked with $\oplus$ or $\ominus$, and runs an off-the-shelf supervised learning algorithm to infer a classifier, which is used to define $\mathcal{F}$.

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

The key idea is to use the impact pre-analysis for Octagon [37], and to convert the results of this pre-analysis to labeled data. Just like the Octagon analysis, this pre-analysis tracks the relationships between variables, but it aggressively abstracts any numerical information so as to achieve better scalability than Octagon. The goal of the pre-analysis is to identify, as much as possible, the case that Octagon would give a precise bound for $\pm x \pm y$, without running Octagon itself. As in Octagon, the pre-analysis computes a matrix with rows and columns for variables with or without the minus sign, but this matrix $m^\sharp$ contains ★ or $\top$, instead of any numerical values. For instance, when applied to our example program, the pre-analysis would infer the following matrix at the first query:

	`a`	`−a`	`b`	`−b`	`c`	`−c`	`i`	`−i`
`a`	★	$\top$	★	$\top$	$\top$	$\top$	★	$\top$
`−a`	$\top$	★	$\top$	★	$\top$	$\top$	$\top$	$\top$
`b`	★	$\top$	★	$\top$	$\top$	$\top$	★	$\top$
`−b`	$\top$	★	$\top$	★	$\top$	$\top$	$\top$	$\top$
`c`	$\top$	$\top$	$\top$	$\top$	★	$\top$	$\top$	$\top$
`−c`	$\top$	$\top$	$\top$	$\top$	$\top$	★	$\top$	$\top$
`i`	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	★	$\top$
`−i`	$\top$	★	$\top$	★	$\top$	$\top$	$\top$	★

(4.3)

Each entry of this matrix stores the pre-analysis's (highly precise on the positive side) prediction on whether Octagon would put a *finite* upper bound at the corresponding entry of its matrix at the same program point. $\bigstar$ means likely, and $\top$ unlikely. For instance, the above matrix contains $\bigstar$ for the entries for $\texttt{i} - \texttt{b}$ and $\texttt{b} - \texttt{a}$, and this means that Octagon is likely to infer finite (thus informative) upper bounds of $\texttt{i} - \texttt{b}$ and $\texttt{b} - \texttt{a}$. In fact, this predication is correct because the actual upper bounds inferred by Octagon are -1 and 0, as can be seen in (4.1).

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

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

Notice that we mark (x, y) with $\oplus$ if tracking the relationship between x and y is useful for some query q'_i. An obvious alternative is to replace some by all, but we found that this alternative led to the worse performance in our experiments.[1] This generation process is applied for all programs $P_1, \ldots, P_N$ in the codebase, and results in the following labeled data: $\mathcal{D} = \bigcup_{1 \le i \le N} \mathcal{D}_{P_i}$. In our example program, if the results of the pre-analysis at both queries are the same matrix in (4.3), our approach picks only the first matrix because the pre-analysis predicts a finite upper

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

bound only for the first query, and it produces the following labeled data from the first matrix:

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

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

Application of an Off-the-shelf Classification Algorithm Once we generate labeled data $\mathcal{D}$, we represent each triple in $\mathcal{D}$ as a vector of $\{0, 1\}$ labeled with $\oplus$ or $\ominus$, and apply an off-the-shelf classification algorithm, such as decision-tree inference [31].

The vector representation of each triple in $\mathcal{D}$ is based on a set of so called features, which are syntactic or semantic properties of a variable pair (x, y) under a program P. Formally, a feature f maps such $(P, (x, y))$ to 0 or 1. For instance, f may check whether the variables x and y appear together in an assignment of the form $x = y + c$ in P, or it may test whether x or y is a global variable. Table 4.1 lists all the features that we designed and used for our variant of the Octagon analysis. Let us denote these features and results of applying them using the following symbols:

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

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

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

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

4.3 Octagon Analysis with Variable Clustering

In this section, we describe a variant of the Octagon analysis that takes not just a program to be analyzed but also clusters of variables in the program. Such clusters are computed according to some strategy before the analysis is run. Given a program and variable clusters, this variant Octagon analysis infers relationships between variables within the same cluster but not across different clusters. Section 5.3.2 presents our method for automatically learning a good strategy for forming such variable clusters.

4.3.1 Programs

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

$$cmd \ ::= \ x = k \mid x = y + k \mid x = ?$$

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

4.3.2 Octagon Analysis

We briefly review the Octagon analysis in [30]. Let $\mathsf{Var}_n = \{x_1, \ldots, x_n\}$ be the set of variables that appear in a program to be analyzed. The analysis aims at finding the upper and lower bounds of expressions of the forms x_i, $x_i + x_j$ and $x_i - x_j$ for

variables $x_i, x_j \in \mathsf{Var}_n$. The analysis represents these bounds as a $(2n \times 2n)$ matrix m of values in $\mathbb{Z} \cup \{\infty\}$, which means the following constraint:

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

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

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

This domain is a complete lattice $(\mathbb{O}, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ where $\top$ is the matrix containing only ∞ and $\sqcup$ and $\sqcap$ are defined pointwise. The details of the lattice structure can be found in [30].

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

The abstract semantics of primitive commands $[\![cmd]\!] : \mathbb{O} \to \mathbb{O}$ is defined in Figure 4.2. The effects of the first two assignments in the concrete semantics can be tracked precisely using abstract elements of Octagon. The abstract semantics of these assignments do such tracking. $[\![x_i = ?]\!]m$ in the last case computes the strong closure of m and forgets any bounds involving x_i in the resulting abstract element $m^\bullet$. The analysis computes a pre-fixpoint of the semantic function $F : (\mathbb{C} \to \mathbb{O}) \to (\mathbb{C} \to \mathbb{O})$ (i.e., X_I with $F(X_I)(c) \sqsubseteq X_I(c)$ for all $c \in \mathbb{C}$):

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

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

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

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

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

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

Figure 4.2: Abstract semantics of some primitive commands in the Octagon analysis. We show the case that the input m is not $\bot$; the abstract semantics always maps $\bot$ to $\bot$. Also, in $x_i = x_j + k$, we consider only the case that $i \neq j$.

4.3.3 Variable Clustering and Partial Octagon Analysis

We use a program analysis that performs the Octagon analysis only partially. This variant of Octagon is similar to those in [30,37]. This partial Octagon analysis takes a collection Π of clusters of variables, which are subsets π of variables in Var_n such that $\bigcup_{\pi \in \Pi} \pi = \mathsf{Var}_n$. Each $\pi \in \Pi$ specifies a variable cluster and instructs the analysis to track relationships between variables in π. Given such a collection Π, the partial Octagon analysis analyzes the program using the complete lattice $(\mathbb{O}_\Pi, \sqsubseteq_\Pi, \bot_\Pi, \top_\Pi, \sqcup_\Pi, \sqcap_\Pi)$ where

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

That is, $\mathbb{O}_\Pi$ consists of families $\{m_\pi\}_{\pi \in \Pi}$ such that each m_π is an abstract element of Octagon used for variables in π, and all lattice operations of $\mathbb{O}_\Pi$ are the

pointwise extensions of those of Octagon. For the example in Section 5.2, if we use $\Pi = \{\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{i}\}\}$, the partial Octagon analysis uses the same domain as Octagon's. But if $\Pi = \{\{\mathtt{a}, \mathtt{b}, \mathtt{i}\}, \{\mathtt{c}\}\}$, the analysis uses the product of two smaller abstract domains, one for $\{\mathtt{a}, \mathtt{b}, \mathtt{i}\}$ and the other for $\{\mathtt{c}\}$.

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

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

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

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

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

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

The abstract semantics of a command updates the component of an input abstract state for a cluster π if the command changes any variable in the cluster; otherwise, it keeps the component. The update is done according to the abstract semantics of Octagon. Notice that the abstract semantics of $x_i = x_j + k$ does not track the relationship $x_j - x_i = k$ in the π component when $x_i \in \pi$ but $x_j \notin \pi$. Giving up such relationships makes this partial analysis perform faster than the original Octagon analysis.

4.4 Learning a Strategy for Clustering Variables

The success of the partial Octagon analysis lies in choosing good clusters of variables for a given program. Ideally each cluster of variable should be relatively small, but if tracking the relationship between variables x_i and x_j is important,

some cluster should contain both x_i and x_j. In this section, we present a method for learning a strategy that chooses such clusters. Our method takes as input a collection of programs, which reflects a typical usage scenario of the partial Octagon analysis. It then automatically learns a strategy from this collection.

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

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

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

4.4.1 Automatic Generation of Labeled Data

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

In order to lessen this scalability issue, we instead run the impact pre-analysis for Octagon from our previous work [37], and convert its results to labeled data. Although this pre-analysis is not as cheap as the Interval analysis, it scales far better than Octagon and enables us to generate labeled data from a wide range of programs. Our learning method then uses the generated data to find a strategy

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

for clustering variables. The found strategy can be viewed as an approximation of this pre-analysis that scales as well as the Interval analysis.

Impact Pre-analysis We review the impact pre-analysis from [37], which aims at quickly predicting the results of running the Octagon analysis on a given program P. Let $n = |\mathsf{Var}_P|$, the number of variables in P. At each program point c of P, the pre-analysis computes a $(2n \times 2n)$ matrix $m^\sharp$ with entries in $\{\bigstar, \top\}$. Intuitively, such a matrix $m^\sharp$ records which entries of the matrix m computed by Octagon are likely to contain nontrivial bounds. If the ij-th entry of $m^\sharp$ is $\bigstar$, the ij-th entry of m is likely to be non-∞ according to the prediction of the pre-analysis. The pre-analysis does not make similar prediction for entries of $m^\sharp$ with $\top$. Such entries should be understood as the absence of information.

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

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

The pre-analysis uses the abstract semantics $[\![cmd]\!]^\sharp : \mathbb{O}^\sharp \to \mathbb{O}^\sharp$ that is derived from this Galois connection and the abstract semantics of the same command in Octagon (Figure 4.2). Figure 4.3 shows the results of this derivation.

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

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

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

Figure 4.3: Abstract semantics of some primitive commands in the impact pre-analysis. In $x_i = x_j + k$, we show only the case that $i \neq j$.

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

$$Q_i' = \{c \mid \exists p.\, (c, p) \in Q_i \land \text{the } jk \text{ entry is about the upper bound claimed in } p \land X_i(c) \neq \bot \land X_i(c)_{jk} = \bigstar\},$$

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

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

4.4.2 Features and Classifier

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

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

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

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

In our experiments, $\mathcal{V}$ has significantly more vectors marked with $\ominus$ than those marked with $\oplus$. We found that the algorithm for inferring a decision tree [31] worked the best for our $\mathcal{V}$

Table 4.1 shows the features that we developed for the Octagon analysis and used in our experiments. These features work for real C programs (not just those in the small language that we have used so far in the paper), and they are all symmetric in the sense that $f_i(P, (x, y)) = f_i(P, (y, x))$. Features 1–6 detect good situations where the Octagon analysis *can* track the relationship between variables *precisely*. For example, $f_1(P, (x, y)) = 1$ when x and y appear in an assignment $x = y + k$ or $y = x + k$ for some constant k in the program P. Note that the abstract semantics of these commands in Octagon do not lose any information. The next features 7–11, on the other hand, detect bad situations where the Octagon analysis *cannot* track the relationship between variables *precisely*. For example, $f_7(P, (x, y)) = 1$ when x or y gets multiplied by a constant different from 1 in a command of P, as in the assignments $y = x * 2$ and $x = y * 2$. Notice that these assignments set up relationships between x and y that can be expressed only approximately by Oc-

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

4.4.3 Strategy for Clustering Variables

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

$$\Pi = \{\pi_1, \dots, \pi_k\} \subseteq \wp(\mathsf{Var}_P),$$

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

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

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

Table 4.1: Features for Relations of Two Variables.

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

4.5 Experiments

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

1. **Effectiveness**: How well does the partial Octagon with a learned strategy perform, compared with the existing Interval and Octagon analyses?
2. **Generalization**: Does the strategy learned from small programs also work well for large unseen programs?
3. **Feature design**: How should we choose a set of features in order to make our method learn a good strategy?
4. **Choice of an off-the-shelf classification algorithm**: Our method uses a classification algorithm for inferring a decision tree by default. How much does this choice matter for the performance of our method?

We conducted our experiments with a realistic static analyzer and open-source C benchmarks. We implemented our method on top of Sparrow, a static buffer-overflow analyzer for real-world C programs [54]. The analyzer performs the combination of the Interval analysis and the pointer analysis based on allocation-site abstraction with several precision-improving techniques such as fully flow-, field-sensitivity and selective context-sensitivity [37]. In our experiments, we modified Sparrow to use the partial Octagon analysis as presented in Section 4.3, instead of Interval. The partial Octagon was implemented on top of the sparse analysis framework [36,37], so it is significantly faster than the vanilla Octagon analysis [30]. For the implementation of data structures and abstract operations for Octagon, we tried the OptOctagons plugin [53] of the Apron framework [20]. For the decision tree learning, we used Scikit-learn [39]. We used 17 open-source benchmark programs (Table 4.3) and all the experiments were done on a Ubuntu machine with Intel Xeon clocked at 2.4GHz cpu and 192GB of main memory.

4.5.1 Effectiveness

We evaluated the effectiveness of a strategy learned by our method on the cost and precision of Octagon. We compared the partial Octagon analysis with a learned variable-clustering strategy with the Interval analysis and the approach for optimizing Octagon in [37]. The approach in [37] also performs the partial Octagon analysis in Section 4.3 but with a fixed variable-clustering strategy that uses the impact pre-analysis online (rather than offline as in our case): the strategy runs the impact pre-analysis on a given program and computes variable clusters of the program based on the results of the pre-analysis. Table 4.3 shows the results of our comparison with 17 open-source programs. We used the leave-one-out cross validation to evaluate our method; for each program P in the table, we applied our method to the other 16 programs, learned a variable-clustering strategy, and ran the partial Octagon on P with this strategy.

The results show that the partial Octagon with a learned strategy strikes the right balance between precision and cost. In total, the Interval analysis reports 7,406 alarms from the benchmark set.[3] The existing approach for partial Octagon [37] reduced the number of alarms by 252, but increased the analysis time by 62x. Meanwhile, our learning-based approach for partial Octagon reduced the number of alarms by 240 while increasing the analysis time by 2x.

We point out that in some programs, the precision of our approach was incomparable with that of the approach in [37]. For instance, for `spell-1.0`, our approach is less precise than that of [37] because some usage patterns of variables in `spell-1.0` do not appear in other programs. On the other hand, for `httptunnel-3.3`, our approach produces better results because the impact pre-analysis of [37] uses $\bigstar$ conservatively and fails to identify some important relationships between variables.

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

Table 4.2: Characteristics of the benchmark programs. **LOC** reports lines of code before preprocessing. **Var** reports the number of program variables (more precisely, abstract locations).

Program	**LOC**	**Var**
brutefir-1.0f	103	54
consolcalculator-1.0	298	165
id3-0.15	512	527
spell-1.0	2,213	450
mp3rename-0.6	2,466	332
irmp3-0.5.3.1	3,797	523
barcode-0.96	4,460	1,738
httptunnel-3.3	6,174	1,622
e2ps-4.34	6,222	1,437
bc-1.06	13,093	1,891
less-382	23,822	3,682
bison-2.5	56,361	14,610
pies-1.2	66,196	9,472
icecast-server-1.3.12	68,564	6,183
raptor-1.4.21	76,378	8,889
dico-2.0	84,333	4,349
lsh-2.0.4	110,898	18,880

4.5.2 Generalization

Although the impact pre-analysis scales far better than Octagon, it is still too expensive to be used routinely for large programs (> 100 KLOC). Therefore, in order for our approach to scale, the variable-clustering strategy learned from a codebase of small programs needs to be effective for large unseen programs. Whether this need is met or not depends on whether our learning method generalize information from small programs to large programs well.

To evaluate this generalization capability of our learning method, we divided the benchmark set into small (< 60 KLOC) and large (> 60 KLOC) programs,

learned a variable-clustering strategy from the group of small programs, and evaluated its performance on that of large programs.

Table 4.4 shows the results. Columns labeled **Small** report the performance of our approach learned from the small programs. **All** reports the performance of the strategy used in Section 4.5.1 (i.e., the strategy learned with all benchmark programs except for each target program). In our experiments, **Small** had the same precision as **All** with negligibly increase in analysis time (4%). These results show that the information learned from small programs is general enough to infer the useful properties about large programs.

4.5.3 Feature Design

We identified top ten features that are most important to learn an effective variable-clustering strategy for Octagon. We applied our method to all the 17 programs so as to learn a decision tree, and measured the relative importance of features by computing their Gini index [5] with the tree. Intuitively, the Gini index shows how much each feature helps a learned decision tree to classify variable pairs as $\oplus$ or $\ominus$. Thus, features with high Gini index are located in the upper part of the tree.

According to the results, the ten most important features are 30, 15, 18, 16, 29, 6, 24, 23, 1, and 21 in Table 4.1. We found that many of the top ten features are negative and describe situations where the precise tracking of variable relationships by Octagon is unnecessary. For instance, feature 30 (size of constant string) and 29 (finite interval) represent variable pairs whose relationships can be precisely captured even with the Interval analysis. Using Octagon for such pairs is overkill. Initially, we conjectured that positive features, which describe situations where the Octagon analysis is effective, would be the most important for learning a good strategy. However, data show that effectively ruling out unnecessary variable relationships is the key to learning a good variable-clustering strategy for Octagon.

4.5.4 Choice of an Off-the-shelf Classification Algorithm

Our learning method uses an off-the-shelf algorithm for inferring a decision tree. In order to see the importance of this default choice, we replaced the decision-tree algorithm by logistic regression [32], which is another popular supervised learning algorithm and infers a linear classifier from labeled data. Such linear classifiers are usually far simpler than nonlinear ones such as a decision tree. We then repeated the leave-one-out cross validation described in Section 4.5.1.

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

4.6 Summary

In this chapter we proposed a method for learning a variable-clustering strategy for the Octagon analysis from a codebase. One notable aspect of our method is that it generates labeled data automatically from a given codebase by running the impact pre-analysis for Octagon [37]. The labeled data are then fed to an off-the-shelf classification algorithm (in particular, decision-tree inference in our implementation), which infers a classifier that can identify important variable pairs from a new unseen program, whose relationships should be tracked by the Octagon analysis. This classifier forms the core of the strategy that is returned by our learning method. Our experiments show that the partial Octagon analysis with the learned strategy scales up to 100KLOC and is 33x faster than the one with the impact pre-analysis

(which itself is significantly faster than the original Octagon analysis), while increasing false alarms by only 2%.

Table 4.3: Comparison of performance of the Interval analysis and two partial Octagon analyses, one with a fixed strategy based on the impact pre-analysis and the other with a learned strategy. **#Alarms** reports the number of buffer-overflow alarms reported by the interval analysis (**Itv**), the partial Octagon analysis with a fixed strategy (**Impt**) and that with a learned strategy (**ML**). **Time** shows the analysis time in seconds, where, in **X**(**Y**), **X** means the total time (including that for clustering and the time for main analysis) and **Y** shows the time spent by the strategy for clustering variables.

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

Table 4.4: Generalization performance.

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

Chapter 5

Selectively Unsound Analysis by Machine Learning

5.1 Introduction

Any realistic bug-finding static analyzers are designed to be unsound. Ideally, a static analyzer is expected to be sound, precise, and scalable; that is, it should be able to consider all program executions and hence do not miss any intended bug while avoiding false positives and scaling to large programs. In reality, however, achieving the three at the same time is extremely challenging, and therefore existing commercial static analysis tools (e.g., [1]) and published static bug-finders (e.g., [12,22,58–60]) trade soundness in order to obtain acceptable performance in precision and scalability.

To our knowledge, all of the existing unsound analysis tools are *uniformly* unsound. For instance, since loops and unknown library calls are major sources of imprecision in static analysis, most static bug-finding tools compromise soundness in analyzing them (e.g., [12,22,58–60]); loops are unrolled for a fixed number of times and subsequent loop iterations are ignored entirely, and unknown library calls are considered as pre-defined behaviors such as skip. All of these approaches are uni-

formly unsound in that they ignore *every* loop and library call in a given program regardless of their different circumstances or properties.

However, this uniform approach to unsoundness has a considerable drawback; it causes the analysis to miss a significant amount of real bugs. For instance, our taint analysis for detecting format-string vulnerabilities ignores the possible data flows of all unknown library calls in the program and therefore only report 5 false alarms in the 13 benchmark C programs (Section 5.5). However, it only managed to detect 16 bugs among the 106 potentially detectable format-string bugs. In other words, this unsound analysis has low false positive rate (FPR $= \frac{\#\text{False Alarms}}{\#\text{All Alarms}}$) but it has high false negative rate (FNR $= \frac{\#\text{Missing Bugs}}{\#\text{All Bugs}}$).

On the other hand, a simple-minded, uniformly sound analysis poses the opposite problem; it has low FNR at the cost of high FPR. For example, a simple solution to decrease the FNR of the unsound taint analysis is to modify the analysis to consider the potential data flows of *every* unknown library call in the program. This uniformly sound analysis is able to find all 106 bugs in the benchmark programs. However, it reports 276 false alarms too.

Our work is to reduce the FNR of an unsound bug-finder while maintaining the original (low) FPR by being *selectively* unsound only when it is likely to be harmless. For example, we unsoundly analyze library calls only when it is likely to reduce FPR while maintaining low FNR. With our approach, the selectively unsound taint analysis reports 92 real bugs (among 106) with 27 false alarms only.

We achieve this by using a machine learning technique that is specialized for anomaly detection [46]. Our key insight is that the program components (e.g., loops and library calls) that produce false alarms are alike, predictable, and sharing some common properties. Meanwhile, the real bugs are often caused by different reasons that are atypical and unpredictable in their own ways (Section 5.3.2) [41]. Based on this observation, we aim to capture the common characteristics of the harmless and precision-decreasing program components by using one-class support vector machines. The entire learning process in our approach (i.e. generating labelled

data and learning a classifier) is fully automatic once a codebase with known bugs is given.

The experimental results show that our method effectively reduces false negatives of the baseline analyzer without sacrificing its precision. We evaluated our method with two realistic static analyzers for C and open-source benchmarks. The first experiment is done with a taint analysis for finding out format-string bugs. In our benchmarks with 106 bugs, the baseline, uniformly unsound analysis detects 16 bugs with 5 false alarms (FPR: 24%, FNR: 85%). Uniformly improving the soundness impairs the precision too much: it reports 106 real bugs with 276 false alarms (FPR: 72%). Our selectively unsound analysis maintains the original precision while greatly decreasing the number of false negatives: it reports 92 bugs with 27 false alarms (FPR: 23%, FNR: 13%). The second experiment is done with an interval analysis for buffer-overflow detection, where we control the soundness for both loops and library calls. In the benchmarks with 138 bugs, the uniformly unsound analysis detects 33 bugs with 104 false alarms (FPR: 76%, FNR: 76%). The uniformly sound analysis detects 118 bugs with 677 false alarms (FPR: 85%). Our selectively unsound analysis detects 96 bugs with 266 false alarms (FPR: 73%, FNR: 30%).

To summarize, our contributions are as follows:

- We present a new approach of selectively employing unsoundness in static analysis. All of the existing bug-finding static analyzers are uniformly unsound.
- We present a machine-learning technique that can automatically tune a static analysis to be selectively unsound. Our technique is based on anomaly detection with automatic generation of labelled data.
- We demonstrate the effectiveness of the technique by experiments with two bug-finding static analyzers for C.

```
str = "hello world'';
for(i=0; str[i]; i++)     // buffer access 1
  skip;

size = positive_input();
for(i=0; i<size; i++)
  skip;

... = str[i];             // buffer access 2
```

Figure 5.1: Example program

5.2 Overview

We illustrate our approach using a static analysis with the interval domain. The goal of the analysis is to detect buffer overflow bugs in a program. For simplicity, we only concern with loops in this section, which could be a potential cause of the buffer overflow bugs.

Consider a simple program in Figure 5.1. In the program, there are two loops and two buffer-access expressions. The first loop iterates over a constant string until the null value in the string is found. In the loop, buffer access 1 is always safe, since `i` is guaranteed to be smaller than the length of `str` inside the loop. On the other hand, buffer access 2 is not always safe, because the index `i` has the value of `size` after the second loop, which can be an arbitrary value due to the external input and may cause a buffer overflow.

5.2.1 Uniformly Unsound Analysis

Consider an analysis that is uniformly unsound for every loop. That is, all the loops in the given program are unrolled for a fixed number of times, and subsequent loop iterations are ignored during the analysis. From the perspective of such an unsound analysis, the example program is treated as follows.

```
str = "hello world";
i = 0;
if (str[i])          // buffer access 1
  skip;

size = positive_input();
i = 0;
if (i < size)
  skip;

... = str[i];        // buffer access 2
```

Note that each loop is unrolled once and replaced with an if-statement. The analysis does not report a false alarm for buffer access 1, since the value of `i` remains as $[0, 0]$. However, it also fails to report a true alarm for buffer access 2; the value of `i` is approximated to $[0, 0]$, hence the analysis considers the buffer access to be safe.

5.2.2 Uniformly Sound Analysis

On the other hand, a sound interval analysis can detect the bug at buffer access 2 with a false alarm at buffer access 1. Inside the first loop, the analysis conservatively approximates the value of `i` to $[0, +\infty]$, since this value is not refined by the loop condition `str[i]`. It is because the interval domain cannot capture non-convex properties (e.g. $\texttt{i} \neq 11$, where 11 is the null index of `str`). Thus, the analysis reports an alarm for buffer access 1 as a potential buffer overflow error, which is a false alarm that we want to avoid. Meanwhile, the variable `i` in the second loop is upper bounded by `size` whose range is approximated as $[0, +\infty]$ due to the unknown input value. Therefore the analyzer reports an alarm for buffer access 2, which is a true alarm in this case.

5.2.3 Selectively Unsound Analysis

Our selectively unsound analyzer applies unsoundness only to the loops that are likely to remove false alarms only. In the example program in Figure 5.1, we ignore the first loop since analyzing it soundly results in reporting a false alarm at buffer access 1. The second loop, on the other hand, needs to be analyzed soundly, since it has the possibility of causing an actual buffer overflow. The selectively unsound analysis on the given program corresponds to analyzing the following program.

```
str = "hello world’’;
i = 0;
if(str[i])         // buffer access 1
  skip;

size = positive_input();
for(i = 0; i < size; i++)
  skip;

... = str[i];      // buffer access 2
```

Note that we only unroll the first loop, not the second loop. By being unsound for the first loop and sound for the second loop, the analysis is able to report the true alarm for buffer access 2 while avoiding the false alarm for buffer access 1.

5.2.4 Our Learning Approach

We achieve the selectively unsound analysis via machine learning-based anomaly detection. Assume that we have a codebase and a set of features. The codebase is a set of programs in which all the bugs are found and their locations are annotated so that we can classify alarms into true or false alarms. Then, we need to decide which set of program components to apply unsoundness selectively. In our example, it is the set of loops in the program we want to analyze. The features in this case describe general characteristics of the loops.

The learning phase consists of three steps.

1. We collect harmless loops from the codebase. A loop is harmless if unsoundly analyzing the loop does not cause to miss real bugs but reduces false alarms. For simplicity, we assume there is only one program in the codebase, and the program contains n loops. When analyzed soundly, it reports certain number of true alarms and false alarms. Then, we examine each loop by replacing it with an if-statement (i.e., unrolling) one by one and compare the result to that of the original program. If the replacement of a loop makes the number of true alarms remain same, but makes the number of false alarms decrease, we consider the loop to be harmless. We collect all the loops satisfying the condition.

2. Next, we represent the loops as feature vectors. Once all the harmless loops in the codebase are collected, we create a feature vector for each loop using the set $f = \{f_1, f_2, \ldots, f_k\}$ where f_i is a predicate over loops. For example, f_1 may indicate whether a loop has a conditional statement containing nulls.

3. Finally, having the generated feature vectors as training data, we learn a classifier that can distinguish such harmless loops. We use one-class classification algorithm [46] for learning the classifier that requires only positive examples (i.e., harmless loops). We use the anomaly detection algorithm to learn the common characteristics and regularities of the harmless loops.

In the testing phase, the classifier takes the feature vectors of all the loops in a new program as an input. If the classifier considers a loop to be harmless, then the loop is analyzed unsoundly, meaning that it is unrolled once and replaced with an if-statement. Otherwise, if the classifier considers a loop to be harmful (i.e., anomaly), then the loop is analyzed soundly.

5.3 Our Technique

Our goal is to find *harmless* components and selectively employ unsoundness only to them. In this section, we describe how to build a selectively unsound static an-

alyzer in detail. First, we introduce a parameterized static analysis that applies unsoundness only to certain program components. Then, we explain how to learn a statistical model from an existing codebase, which is used to derive a soundness parameter.

5.3.1 Parameterized Static Analysis

Our analysis employs a parameterized strategy for selecting the set of program components that will be analyzed soundly. This is a variant of the well-known setting for the parameterized static analysis [26, 38], except the parameter controls the soundness of the analysis, not the precision.

Let $P \in \mathit{Pgm}$ be a program that we want to analyze. $\mathbb{C}_P$ is the set of program points in P. $\mathbb{J}_P$ is the set of program components such as the set of loops, the set of library calls, or the set of other operations in P. In the rest of this section, we omit the subscript P from $\mathbb{C}_P$ and $\mathbb{J}_P$ when there is no confusion.

The selectively unsound static analyzer is a function

$$F : \mathit{Pgm} \times \wp(\mathbb{J}) \to \wp(\mathbb{C})$$

which is parameterized by the soundness parameter $\pi \in \wp(\mathbb{J})$ (i.e. a set of program components). Given a program P and its parameter π, the analyzer outputs alarms (i.e. a set of program points).

A soundness parameter $\pi \in \wp(\mathbb{J})$ is a set of program components which need to be analyzed soundly. In other words, it selects the program components that are likely to produce true alarms as a result of detecting real bugs in the program. For instance, when $\mathbb{J} = \{j_1, \cdots, j_n\}$ is the set of loops in the program P, $j_i \in \pi$ means that the ith loop in the program is not considered to be harmless; we analyze the loop as it is rather than unrolling the loop once and ignoring all the subsequent loop iterations.

We want to find a good soundness parameter which allows the analyzer to apply costly soundness only to the necessary components which are not harmless. Let **1** be the parameter where every component is selected and **0** be the param-

eter where no component is selected. Then, $F(P, \mathbf{1})$ denotes the analysis that is fully sound, which can detect the maximum number of the real bugs along with lots of false alarms. $F(P, \mathbf{0})$ means the fully unsound analysis, reporting the minimum number of false alarms with risk of missing many real bugs. For our analysis, it is important to find a proper parameter which strikes the balance between **1** and **0**, reporting false alarms as few as possible while detecting most of the real bugs.

5.3.2 Learning a Classifier

We want to build a classifier which can predict whether a given program component is harmless or not. The classifier in our approach exploits general properties of harmless components and uses the information for new, unseen programs.

Features

We define features to capture common properties of program components. Features are either syntactic or semantic properties of program components, which have either binary or numeric values. For simplicity, we assume them to be binary properties: $f_i : \mathbb{J} \to \{0, 1\}$. Given a set of features, we can derive a feature vector for each program component. Suppose that we have n features: $f = \{f_1, \ldots, f_n\}$. With the set of features, each program component $j \in \mathbb{J}$ can be represented as a feature vector $f(j) = \langle f_1(j), \ldots, f_n(j) \rangle$.

Our approach requires analysis designers to come up with a set of features for each parameterized static analysis F. In Section 5.4, we discuss how to construct program features with two case studies for loops and library calls.

Learning Process

A classifier is defined as a function

$$\mathcal{C} : \{0, 1\}^n \to \{0, 1\}$$

which takes a feature vector of a program component as an input. It returns 1 if it considers the component to be harmless or 0, otherwise.

We define a model $\mathcal{M} : Pgm \to \wp(\mathbb{J})$ that is used to derive a soundness parameter for a given program as follows:

$$\mathcal{M}(P) = \{j \in \mathbb{J} \mid \mathcal{C}(f(j)) = 0\}.$$

The model collects the program components that are potentially harmful, which may cause real bugs. With the model, we run the static analysis for a new, unseen program P as follows:

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

That is, we first obtain the soundness parameter $\mathcal{M}(P)$ from the model and instantiate the static analysis with the parameter. As a result, the analysis becomes sound for the program components that are selected by the parameter from the model and unsound for the others.

We learn the model with One-Class Support Vector Machine (OC-SVM) [46]. OC-SVM is an unsupervised algorithm that learns a model for anomaly detection: classifying new data as similar or different to its training data. Our intuition is that harmless program components tend to be typical, sharing common properties, whereas harmful components are atypical, therefore difficult to be characterized. It is because bugs in the real world are introduced unexpectedly by nature. In addition, collecting examples for all kinds of bugs is infeasible, whereas collecting and generalizing the characteristics of harmless components is relatively easy to achieve. Therefore, we use this one-class classification method; it only requires positive examples (e.g., harmless loops) that are expected to share some regularities, learns such regularities, and classifies new data as similar or different to its training data.

Note that the characteristics of harmless components are largely determined by the design choices of a given static analysis (e.g., abstract domain), whereas that

of harmful components are not affected by the analysis design. For example, for an interval analysis of C programs, the following loops are typically harmless:

- Loops iterating over constant strings:

```
str=‘‘hello world’’;
for(i=0; str[i]; i++) // false alarm
  ...
```

  As explained before, analyzing such loops soundly is likely to cause false alarms, rather than detecting true bugs, because of the non-disjunctive limitation of the interval domain.

- Loops involving variable relationships:

```
p=malloc(len);
for (i = 0; i < len; i++)
  p[i] = ... // false alarm
```

  Sound analysis of this kind of loops is likely to produce false alarms because of the non-relational limitation of the interval domain. The analysis cannot track the relationship between the value of `len`, the value of `i`, and the size of buffer `p`.

Generating Training Data

From an existing codebase, we generate training data for learning the classifier. The training dataset is composed of a set of feature vectors. Note that we only collect the feature vectors of harmless components, because OC-SVM is designed to learn the regularities of positive examples. The positive examples in our case are the harmless components.

The codebase of the system is a set of annotated programs

$$\mathbf{P} = \{(P_1, B_1), \ldots, (P_n, B_n)\},$$

Algorithm 1 Training data generation

```
1: T := ∅
2: for all (P_i, B_i) ∈ P do
3:    A_i = F(P_i, 1)
4:    (A_t, A_f) := (A_i ∩ B_i, A_i \ B_i)
5:    for all j ∈ 𝕁_{P_i} do
6:       A'_i = F(P_i, 1 \ {j})
7:       (A'_t, A'_f) = (A'_i ∩ B_i, A'_i \ B_i)
8:       if |A'_t| = |A_t| ∧ |A'_f| < |A_f| then
9:          T := T ∪ {f(j)}
10:      end if
11:   end for
12: end for
```

in which each program P_i is associated with a set of buggy program points $B_i \subseteq \mathbb{C}_{P_i}$. Once all the programs in the codebase is annotated accordingly, we can automatically generate training data for the classifier. We first applies unsoundness to each component one by one, runs the analysis, and collects the feature vectors from all the harmless components in the given codebase. We consider a program component to be harmless if the number of true alarms remains same and the number of false alarms is decreased, when analyzed unsoundly.

The algorithm for generating training data is shown in Algorithm 1. For each program P_i in the codebase, we run the fully sound static analysis and classify the output alarms A_i into true alarms A_t and false alarms A_f (line 3 and 4). Then, for each program component $j \in \mathbb{J}_{P_i}$, we run the static analysis without the jth component (i.e. $\mathbf{1} \setminus \{j\}$) (line 6). The component j is considered to be harmless when the analysis which is unsound for j still captures all the real bugs (i.e. $|A'_t| = |A_t|$) but reports fewer false alarms (i.e. $|A'_f| < |A_f|$) compared to the fully sound analysis (line 8). We collect feature vectors from all the harmless program components into the training set $\mathbf{T} \subseteq \{0, 1\}^n$.

5.4 Instance Analyses

In this section, we present a generic static analysis that is selectively unsound for loops and library calls as well as a set of features for them. We have chosen loops and library calls because they are the main sources of false alarms from real-world static analyzers and thus often made unsound in practice (e.g. [12, 22, 58–60]). In the analysis, loops are unrolled for a fixed number of times and library calls are simply ignored as skips. Our aim is to selectively unroll and ignore loops and library calls, respectively, only when doing so is harmless.

We present two instances of the analysis, one for an interval analysis and the other for a taint analysis. The interval analysis is used to find out possible buffer-overflow errors, and the taint analysis is for detecting format string vulnerabilities (i.e. uses of unchecked user input as format string parameters of certain C functions such as `printf`). The soundness of these instance analyses is tuned by our technique (Section 5.3), where we used the same set of features designed for the generic analysis.

In Section 5.4.1, we define the generic analysis with features. Section 5.4.2 and Section 5.4.3 present two instances, namely the interval analysis and the taint analysis.

5.4.1 A Generic, Selectively Unsound Static Analysis

Abstract Semantics

We consider a family of static analyses whose soundness is parametric for loops and library calls. Consider the following simple imperative language:

$$
\begin{array}{rcl}
C & \rightarrow & L := E \mid C_1; C_2 \mid \texttt{if}\ E\ C_1\ C_2 \\
 & \mid & \texttt{while}_l\ E\ C \mid L := lib_l() \\
E & \rightarrow & n \mid L \mid \texttt{alloc}_l(E) \mid \&L \mid E_1 + E_2 \\
L & \rightarrow & x \mid *E \mid E_1[E_2]
\end{array}
$$

$$
\begin{array}{rcl}
F_\pi(L := E, s) & = & s[\mathcal{L}(L, s) \mapsto \mathcal{V}(E, s)] \\
F_\pi(C_1; C_2, s) & = & F_\pi(C_2, F_\pi(C_1, s)) \\
F_\pi(\texttt{if}\ E\ C_1\ C_2, s) & = & F_\pi(C_1) \sqcup F_\pi(C_2) \\
F_\pi(\texttt{while}_l\ E\ C, s) & = & \begin{cases} \mathsf{fix}(\lambda X.s \sqcup F_\pi(C, X)) & \text{if } l \in \pi \\ F_\pi(C, s) & \text{otherwise} \end{cases} \\
F_\pi(L := lib_l(), s) & = & \begin{cases} s[\mathcal{L}(L, s) \mapsto \top] & \text{if } l \in \pi \\ s & \text{otherwise} \end{cases}
\end{array}
$$

Figure 5.2: Static analysis selectively unsound for loops and library calls

A command is an assignment, sequence, if statement, while statement, or a call to an unknown library function. In the program, loops and library calls are labelled and the set of labels forms $\mathbb{J}$ in Section 5.3.1. The parameter space is the set of all subsets of program labels, i.e., $\wp(\mathbb{J})$. We assume that labels in the program are all distinct. An expression is an integer (n), l-value expression (L), array allocation ($\texttt{alloc}_l(E)$) where E is the size of the array to be allocated and l is the label for the allocation site, address-of expression ($\&L$), or compound expression ($E + E$). An l-value expression is a variable (x) or array access expression ($E_1[E_2]$).

The abstract semantics of the analysis is defined in Figure 5.2. The analysis is parameterized by $\pi \in \wp(\mathbb{J})$, a set of labels, and is unsound for loops and library calls not included in π. The abstract semantics is defined by the semantic function $F_\pi : C \times \mathbb{S} \to \mathbb{S}$, where $\mathbb{S}$ is the domain of abstract states mapping abstract locations to abstract values, i.e., $\mathbb{S} = \mathbb{L} \to \mathbb{V}$. The analysis is generic in that abstract locations ($\mathbb{L}$) and values ($\mathbb{V}$) are unspecified. They will be given for each analysis instance in subsequent subsections. We assume that the abstract domain is accompanied by two functions $\mathcal{L} : L \times \mathbb{S} \to \wp(\mathbb{L})$ and $\mathcal{V} : E \times \mathbb{S} \to \mathbb{V}$, which compute abstract locations and values of given l-value and r-value expressions, respectively.

The abstract semantics is standard except for the selective treatment of soundness. For a loop statement ($\text{while}_l\ E\ C$), the analysis applies the usual (sound) fixed point computation (fix is a pre-fixpoint operator) when the label l is included in the parameter π. When a loop is not included in π, the analysis ignores the loop

$$
\begin{aligned}
\mathbb{L} &= \mathit{Var} + \mathit{AllocSite} \\
\mathbb{V} &= \mathbb{I} \times \wp(\mathbb{L}) \times \wp(\mathbb{A}) \\
\mathbb{I} &= \{\bot\} \cup \{[l,u] \mid l,u \in \mathbb{Z} \cup \{\pm\infty\}\} \\
\mathbb{A} &= \mathbb{L} \times \mathbb{I} \times \mathbb{I}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{L}(x,s) &= \{x\} \\
\mathcal{L}(*E,s) &= \mathcal{V}(E,s).2 \\
\mathcal{L}(E_1[E_2],s) &= \{a \mid \langle a, _, _\rangle \in \mathcal{V}(E_1,s).3\}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}(n,s) &= \langle [n,n], \emptyset, \emptyset \rangle \\
\mathcal{V}(L,s) &= \bigsqcup \{s(l) \mid l \in \mathcal{L}(L,s)\} \\
\mathcal{V}(\mathtt{alloc}_l(E),s) &= \langle \bot, \{l\}, \{\langle l, [0,0], \mathcal{V}(E,s).1\rangle\}\rangle \\
\mathcal{V}(\&L,s) &= \langle \bot, \mathcal{L}(L,s), \emptyset \rangle \\
\mathcal{V}(E_1 + E_2,s) &= \mathcal{V}(E_1,s) \mathbin{\hat{+}} \mathcal{V}(E_2,s)
\end{aligned}
$$

Figure 5.3: Abstract domain and semantics for interval analysis

and execute the body C only once (i.e. unrolling the loop once). For unknown library calls, the analysis conservatively updates the return location L when l is chosen, i.e., $l \in \pi$. Otherwise, we completely ignore the effect of the library call. Thus, π determines how soundly we analyze the program with respect to loops and unknown library calls. For instance, when $\pi = \mathbb{J}$, the analysis is maximally conservative for loops and library calls, and when $\pi = \emptyset$, the analysis is completely unsound and ignores all of the loops and library calls in the program.

Features

We have designed a set of features for loops and library calls, which can be used for instantiating the generic analysis above. We examined open-source C programs and identified 37 features (Table 5.1 and 5.2) that describe common characteristics of loops and library calls in typical C programs.

The features are classified into syntactic and semantic features. A syntactic feature describes a property that can be checked by a simple syntax analysis. For ex-

$$
\begin{array}{rcl}
\mathbb{L} & = & \mathit{Var} + \mathit{AllocSite} \\
\mathbb{V} & = & \{\bot, \top\} \times \wp(\mathbb{L}) \\
\\
\mathcal{L}(x, s) & = & \{x\} \\
\mathcal{L}(E_1[E_2], s) & = & \mathcal{V}(E_1, s).2 \\
\mathcal{V}(n, s) & = & \begin{cases} \langle \top, \emptyset \rangle & \text{if } n \in \mathbb{T} \\ \langle \bot, \emptyset \rangle & \text{otherwise} \end{cases} \\
\mathcal{V}(L, s) & = & \bigsqcup \{ s(l) \mid l \in \mathcal{L}(L, s) \} \\
\mathcal{V}(\texttt{alloc}_l(E), s) & = & \langle \bot, \{l\} \rangle \\
\mathcal{V}(\&L, s) & = & \langle \bot, \mathcal{L}(L, s) \rangle \\
\mathcal{V}(E_1 + E_2, s) & = & \mathcal{V}(E_1, s) \sqcup \mathcal{V}(E_2, s)
\end{array}
$$

Figure 5.4: Abstract domain and semantics for taint analysis

ample, a syntactic feature characterizes loops whose conditional expressions involve constant values, or library calls whose return type is an integer. A semantic feature describes a property that requires a (yet simple) data-flow analysis. For instance, a semantic feature for loops describes that the loop condition involves an expression whose value depends on some external input of the program:

```
c = input();  // external input
b = c;
while (a < b) { ... }
```

To figure out that the value of `b` comes from the external input, we need to track the data-flow of the external value. Each feature is either binary or numeric, where all the numeric features are normalized to a real number between 0 and 1 based on relative quantities within a single program.

We designed those features with generality in mind so that the features can be reused for different analyses as much as possible. Note that the features in Table 5.1 and 5.2 are not dependent on a particular static analysis, but describe rather general, syntactic and semantic program properties. We use the same set of fea-

tures for the interval and taint analyses and show that we can effectively tune the soundness of both analyses with the single set of features as shown in Section 5.5.

5.4.2 Instantiation 1: Interval Analysis

We first instantiate the generic analysis with the interval domain and use it to find out potential buffer-overflow errors in the program.

The generic analysis left out the definitions of abstract locations ($\mathbb{L}$), abstract values ($\mathbb{V}$), and the evaluation functions for them ($\mathcal{L}$ and $\mathcal{V}$). These definitions for the interval analysis are given in Figure 5.3. An abstract location is either a variable or an allocation-site. An abstract value is a tuple of an interval ($\mathbb{I}$), which is an abstraction of set of numeric values, a points-to set ($\wp(\mathbb{L})$) and a set of abstract arrays ($\wp(\mathbb{A})$). Abstract array $\langle a, o, s \rangle$ has the abstract location($a \in \mathbb{L}$), offset ($o \in \mathbb{I}$), and size ($s \in \mathbb{I}$). The evaluation function $\mathcal{L}$ takes an l-value expression and an abstract state, and computes the set of abstract locations that the l-value denotes. The function $\mathcal{V}(E, s)$ evaluates to the abstract value of E under s. In the definition, we write $\mathcal{V}(E, s).n$ for the nth component of the abstract value of $\mathcal{V}(E)$.

The analysis reports a buffer-overflow alarm when the index of an array can be greater than its size according to the analysis results. For example, consider an expression `arr[idx]`. Suppose the analysis concludes that `arr` has an array of $\langle l, [0, 0], [5, 10] \rangle$ (i.e. an array of size $[5, 10]$) and the interval value of `idx` is $[3, 7]$. The analysis raises an alarm at the array expression because the index value may exceed the size of the array (e.g. when the size is 5 and the index is 7).

5.4.3 Instantiation 2: Taint Analysis

The second instance is a taint analysis for detecting format string vulnerabilities in C programs. The abstract domain and semantics are given in Figure 5.4. The analysis combines a taint analysis and a pointer analysis, and therefore an abstract location is still either a variable or an allocation-site. An abstract value is a tuple of a taint value and a points-to set. The taint domain consists of two abstract values: $\top$ is used to indicate that the value is tainted and $\bot$ represents untainted

values. For simplicity, we model taint sources by a particular set $\mathbb{T} \subseteq \mathbb{Z}$ of integers; constant integer n generates a taint value $\top$ if $n \in \mathbb{T}$. In actual implementation, $\top$ is produced by function calls that receives user input such as `fgets`. The analysis reports an alarm whenever a taint value is involved in a format string parameter of functions.

5.5 Experiments

We empirically show the effectiveness of our approach on selectively applying unsoundness only to harmless program components. We design the experiments to address the following questions:

- **Effectiveness of Our Approach**: How much is the selectively unsound analysis better than the fully sound or fully unsound analyses?
- **Efficacy of OC-SVM**: Does the one-class classification algorithm outperform two-class classification algorithms?
- **Feature Design**: How should we choose a set of features to effectively predict harmless program components?
- **Time Cost**: How does our technique affect cost of analysis?

5.5.1 Setting

Implementation

We have implemented our method on top of a static analyzer for full C. It is a generic analyzer that tracks all of numeric, pointer, array, and string values with flow-, field-, and context-sensitivity. The baseline analyzer is unsound by design to achieve a precise bug-finder; it ignores complex operations (e.g., bitwise operations and weak updates) and filters out reported alarms that are unlikely to be true.

We modified the baseline analyzer and created two instance analyzers, an interval analysis and a taint analysis, as described in Section 5.4. For each analysis,

we built a fully sound version (Sound), a uniformly unsound version (Uniform), and a selectively unsound version (Selective) with respect to the soundness parameter in Section 5.4. In the interval analysis for buffer-overflow errors, Uniform is set to be uniformly unsound for every loop and library call, and Selective is selectively unsound for them. In the taint analysis for format string vulnerabilities, Uniform is uniformly unsound for all the library calls (but not for loops), and Selective is selectively unsound for them.

To implement the OC-SVM classifier, we used scikit-learn machine-learning package [47] with the default setting of the algorithm (specifically, we used the radial basis function (RBF) kernel with $\gamma = 0.1$ and $\nu = 0.1$).

Benchmark

Our experiments were performed on 36 programs whose buggy program points are known. They are the programs from open source software packages or previous work on static analysis evaluations [27, 63]. Table 5.3 and 5.4 contain the list of the benchmark programs for the interval and the taint analysis, respectively. SM-X, BIND-X, and FTP-X are model programs from [63], which contain buffer overflow vulnerabilities. Most of the bugs in the benchmarks are reported as critical vulnerabilities by authorities such as CVE [7]. In total, our benchmark programs have 138 real buffer-overflow bugs and 106 real format string bugs.

5.5.2 Effectiveness of Our Approach

We evaluate the effectiveness of our approach by comparing precision of Selective to that of the other analyzers, Sound and Uniform. We use cross-validation, a model validation technique for assessing how the results of a statistical analysis will generalize to new data. We show the results from three types of cross-validation: leave-one-out, 2-fold, and 3-fold cross-validation.

Leave-one-out Cross-validation

This is one of the most common types of cross-validation, which uses one observation as the validation set and the remaining observations as the training set. In case of the interval analysis, for example, among the 23 benchmark programs, one program is used for validating and measuring the effectiveness of the learned model, and the other remaining 22 programs are used for training.

Table 5.3 shows the results of the leave-one-out cross-validation for the interval analysis. We measured the number of true (**T**) and false (**F**) alarms from Sound, Uniform, and Selective. In terms of true alarms, Sound detects 118 real bugs (FNR: 14.5%) in the programs. While Uniform detects only 33 bugs (FNR: 76.1%), Selective effectively detects 100 bugs (FNR: 27.5%). Meanwhile, Sound reports 677 false alarms (FPR: 85.2%).[1] Uniform, on the other hand, reports 104 false alarms (FPR : 75.9%), which indicates 573 alarms can be potentially removable by being unsound for loops and library calls. Among the 573 alarms, Selective can remove 72.1% (413/573) of them (FPR:72.5%).

Table 5.4 shows the results for the taint analysis. In total, Sound detects all of the 106 real format-string bugs in the programs, while Uniform detects only 16 bugs (FNR: 84.9%). On the contrary, Selective effectively detects 92 bugs (FNR: 13.2%). Meanwhile, Sound, Uniform, and Selective report 273, 5, and 28 false alarms, respectively. That is, among 273 false alarms, which can be potentially removable by being unsound for library calls, Selective can remove 89.7% (245/273) of them.

The result implies that selectively applying unsoundness is also crucial for reducing FPR of the analysis. For the interval analysis, the FPR is 85.2% for Sound and 75.9% for Uniform, whereas 72.5% for Selective on average. For the taint analysis, the FPR is 72.0% for Sound, 23.3% for Selective, 23.8% for Uniform on average.

[1] In practice, eliminating these false alarms is extremely challenging in domain-unaware static analysis, because they arise from a variety of reasons: e.g., large recursive call cycles, unknown library calls, complex loops, heap abstractions, etc.

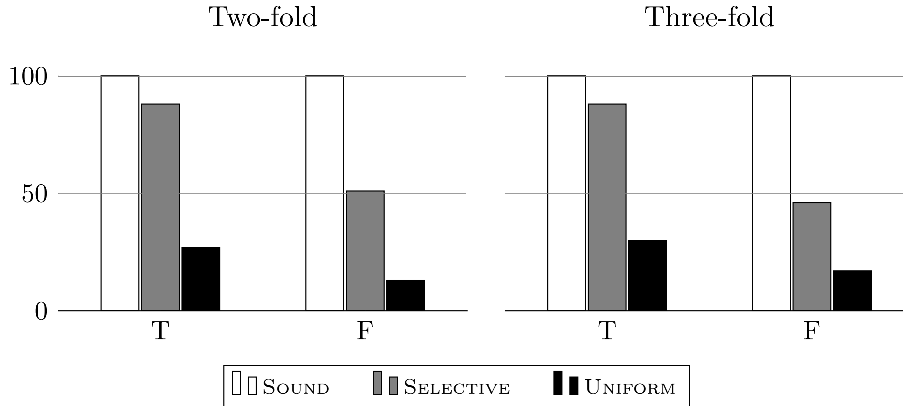


Figure 5.5: Performance with different training and test data

Two- and Three-fold Cross-validation

Next, we evaluate the performance of the interval analysis with 2-fold and 3-fold cross-validation. The benchmark is randomly divided into 2 or 3 subsets that are equal size. Then, one of them is used as the validation set and the others as the training sets. We repeated this process ten times and reported the number of alarms for each trial.

Figure 5.5 shows the number of true and false alarms for each trial of 2-fold and 3-fold cross-validation. The numbers are normalized with respect to the number of alarms produced by Sound. In total, Sound reported 486 true alarms and 3,696 false alarms. Selective detected 427 (87.9%) true alarms, whereas Uniform detected only 129 (26.5%) true alarms in the 2-fold cross-validation. Compared to Sound, Selective reduced 1,812 (49.0%) false alarms, while Uniform reduced 3,216 (87.0%). During the 3-fold cross-validation, Sound reported 399 true alarms and 2,119 false alarms. In terms of true alarms, Selective detected 352 (88.2%) true alarms, whereas Uniform managed to detect only 119 (29.8%) true alarms. As for false alarms, among 1,769 (83.5%) false alarms that are reduced by Uniform, Selective was able to reduce 1,150 (54.3%).

5.5.3 Efficacy of OC-SVM

In this section, we justify the use of OC-SVM for learning common properties of harmless program components. We compare the performance of SELECTIVE whose classifier is learned by OC-SVM to that of three other analyzers with a binary classifier and two random classifiers, respectively.

Let BINARY be the analyzer with a binary classifier. We use C-SVM for the binary classifier, which is a support vector machine-based binary classification algorithm [3]. It learns two classes of training data (i.e. a set of harmless components and the complement set), and then decides whether a new input is harmless or not. In these experiments, we used the interval analyzer with leave-one-out cross validation.

RANDA and RANDB are the analyzers with random classifiers that are built and used for the comparison. RANDA randomly classifies components as harmless with the probability of 0.5. Stochastically, a half of loops and library calls are selected as harmless. RANDB randomly classifies components as harmless with the same probability of the OC-SVM. We ran each analyzer 10 times and measured the number of alarms for each trial.

Figure 5.6 compares the number of true and false alarms produced by SELECTIVE, BINARY, RANDA, and RANDB for 10 trials. BINARY reports more true alarms than SELECTIVE; BINARY reports 103 true alarms, whereas SELECTIVE reports 96 true alarms. However, using BINARY considerably sacrifices the precision; it reports 573 false alarms, whereas SELECTIVE reports only 266. The results from RANDA and RANDB are also inferior to SELECTIVE; RANDA reports 387.5 false alarms and 70.5 true alarms, and RANDB reports 267.2 false alarms with 79.4 true alarms on average.

The result shows SELECTIVE clearly outperforms the other classifiers. SELECTIVE is more precise than BINARY, indicating that the anomaly detection by OC-SVM is more suitable to find harmless components than the binary classification. Also, SELECTIVE always detects more bugs and reports less false alarms than other analyzers with the random classifiers. Despite the fact that RANDB detects more

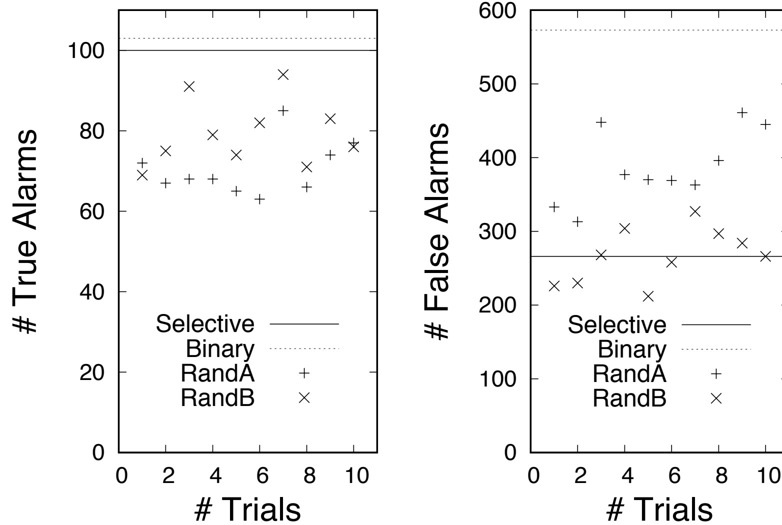


Figure 5.6: Comparison between Selective, Binary, RandA, and RandB

bugs than RandA, it is still insignificant since both of them are much more imprecise than our system.

5.5.4 Feature Design

Wining Features

The learned classifier tells us which feature is most useful for learning harmless unsoundness. The features we used capture general characteristics of harmless program components. In order to determine the ordering of features, we used information gain which is the expected reduction in entropy when a feature is used to classify training examples (in classification, low entropy, i.e., impure data, is preferred) [31].

The results show that harmless loops tend to have pointers iteratively accessing (PtrInc) arrays (Array) or strings (FinString) with loop conditions that compare array contents with null (Null) or constant values (Const). These features collectively describe loops like the first example loop in Section 5.2. The result also

shows that most harmless library calls for the interval analysis return integer values (Int) and manipulate strings (CString). This is because our interval analyzer aggressively abstracts string values, so unsound treatment of string libraries (e.g., `strlen`, `strcat`) is likely to improve the analysis precision. For the taint analysis, the results show that library calls with less arguments (#Args) and abstract locations (#AbsLoc) (e.g., `random`, `strlen`) are likely to be irrelevant to propagation of user inputs compared to ones with more arguments (e.g., `fread`, `recv`).

Different Feature Sets

We measured the performance of the classifier with less features in three ways: 1) with syntactic features only; 2) with semantic features only; and 3) with randomly-chosen half of the features. For the interval analyzer, the classifier learned with only syntactic features reported 1% more bugs but 26% more false alarms than the classifier with all features, the classifier with only semantic features reported 1% more false alarms and missed 41% more bugs, and the classifier with half of the features reported 17% more false alarms and missed 1% more bugs on average.

5.5.5 Time Cost

We measured how long it takes to run each analysis on our benchmark programs and compare it with the time that our selective unsound analysis takes. For the benchmark programs in Table 5.3, the sound interval analysis Sound took 42.1 seconds for analyzing all the listed programs, Uniform only took 27.7 seconds, reducing the total time by 14.4 seconds (34.2%). Selective took 33.8 seconds, reducing the total time by 8.3 seconds (19.7%). RandA and RandB took longer than Selective: 35.4 and 37.5 seconds, respectively. In summary, Selective takes less time than Sound, RandA, and RandB.

5.5.6 Discussion

As addressed in the experiments, our technique may miss some true alarms which can be detected by the fully sound analysis or fail to avoid some false alarms which

are not reported by the fully unsound analysis. In this section, we discuss why these limitations occur and how to overcome.

Remaining False Alarms

Compared to the fully unsound analysis, our technique reports more false alarms. It is mainly because reporting the false alarms is inevitable in order to detect true alarms. Consider the following example program excerpted from SM-5:

```
1 size = 10 + positive_input();
2 arr = malloc(size);
3
4 for(i = 0; i < size; i++){
5   arr[i] = ...          // buffer access 1
6   arr[i+1] = ...         // buffer access 2
7 }
```

By soundly analyzing the loop, the analysis reports an alarm for the buffer-overflow bug at line 6 at the cost of a false alarm at line 5. The unsound analysis removes the false alarm, but it also fails to report the true alarm. Our selective method may decide to analyze such a loop soundly in order to detect the bug, even though reporting the false alarm is inevitable.

We found that these inevitable false alarms are the primary reason for SELECTIVE to report more false alarms compared to UNIFORM. For example, when analyzing SM-5 in our benchmark programs, five among six false alarms are inevitable. In order to remove such false alarms in a harmless way, we need a more fine-grained parameter space for soundness so that we can apply different degrees of soundness to different statements in a single loop, which would be an interesting future direction to investigate.

Missing True Alarms

Compared to the fully sound analysis, our technique reports less true alarms. It is mainly because the bugs are involved in typically-harmless loops. Consider the following code snippet from man-1.5h1:

```
1 char arr[10] = ‘‘string’’;
2 size = positive_input();
3 for (i = 0; i < size; i ++)
4   skip;
5 arr[i] = 0;              // buffer access 1
6
7 for(i = 0; arr[i]; i++)  // buffer access 2
8   skip;
```

The two buffer access expressions both contain buffer overflow bugs. However, our technique detects the first bug, but not the second. It is because it classifies the second loop as harmless–it learns that loops that iterate constant strings are likely to be harmless.

However, we found that most of the missing bugs share the root causes with other bugs detected by our technique. For instance, in the above example, fixing the first bug at line 5 automatically fixes the second one. In our case, therefore, missing true alarms is in fact not a huge drawback in terms of practicability.

5.6 Summary

In this chapter, we presented a novel approach for employing unsoundness in static analysis. Unlike existing uniformly unsound analyses, our technique selectively applies unsoundness only when it is likely to be harmless (i.e. reducing the number of false alarms while retaining true alarms). We proposed a learning-based method for automatically tuning the soundness of static analysis in such a harmless way. The experimental results showed that the technique is effectively applicable to two

bug-finding static analyzers and reduces their false negative rates while retaining their original precision.

Target	Feature	Property	Type	Description
Loop	Null	Syntactic	Binary	Whether the loop condition contains nulls or not
	Const	Syntactic	Binary	Whether the loop condition contains constants or not
	Array	Syntactic	Binary	Whether the loop condition contains array accesses or not
	Conjunction	Syntactic	Binary	Whether the loop condition contains `&&` or not
	IdxSingle	Syntactic	Binary	Whether the loop condition contains an index for a single array in the loop
	IdxMulti	Syntactic	Binary	Whether the loop condition contains an index for multiple arrays in the loop
	IdxOutside	Syntactic	Binary	Whether the loop condition contains an index for an array outside of the loop
	InitIdx	Syntactic	Binary	Whether an index is initialized before the loop
	Exit	Syntactic	Numeric	The (normalized) number of exits in the loop
	Size	Syntactic	Numeric	The (normalized) size of the loop
	ArrayAccess	Syntactic	Numeric	The (normalized) number of array accesses in the loop
	ArithInc	Syntactic	Numeric	The (normalized) number of arithmetic increments in the loop
	PointerInc	Syntactic	Numeric	The (normalized) number of pointer increments in the loop
	Prune	Semantic	Binary	Whether the loop condition prunes the abstract state or not
	Input	Semantic	Binary	Whether the loop condition is determined by external inputs
	GVar	Semantic	Binary	Whether global variables are accessed in the loop condition
	FinInterval	Semantic	Binary	Whether a variable has a finite interval value in the loop condition
	FinArray	Semantic	Binary	Whether a variable has a finite size of array in the loop condition
	FinString	Semantic	Binary	Whether a variable has a finite string in the loop condition
	LCSize	Semantic	Binary	Whether a variable has an array of which the size is a left-closed interval
	LCOffset	Semantic	Binary	Whether a variable has an array of which the offset is a left-closed interval
	#AbsLoc	Semantic	Numeric	The (normalized) number of abstract locations accessed in the loop

Table 5.1: Features for typical loops in C programs

Target	Feature	Property	Type	Description
Library	Const	Syntactic	Binary	Whether the parameters contain constants or not
	Void	Syntactic	Binary	Whether the return type is void or not
	Int	Syntactic	Binary	Whether the return type is int or not
	CString	Syntactic	Binary	Whether the function is declared in `string.h` or not
	InsideLoop	Syntactic	Binary	Whether the function is called in a loop or not
	#Args	Syntactic	Numeric	The (normalized) number of arguments
	DefParam	Semantic	Binary	Whether a parameter are defined in a loop or not
	UseRet	Semantic	Binary	Whether the return value is used in a loop or not
	UptParam	Semantic	Binary	Whether a parameter is update via the library call
	Escape	Semantic	Binary	Whether the return value escapes the caller
	GVar	Semantic	Binary	Whether a parameters points to a global variable
	Input	Semantic	Binary	Whether a parameters are determined by external inputs
	FinInterval	Semantic	Binary	Whether a parameter have a finite interval value
	#AbsLoc	Semantic	Numeric	The (normalized) number of abstract locations accessed in the arguments
	#ArgString	Semantic	Numeric	The (normalized) number of string arguments

Table 5.2: Features for typical library calls in C programs

Table 5.3: The number of alarms in interval analysis

Program	LOC	Bug	SOUND		SELECTIVE		UNIFORM	
			T	**F**	**T**	**F**	**T**	**F**
SM-1	0.5K	28	28	18	28	15	13	5
SM-2	0.8K	2	2	16	1	4	0	0
SM-3	0.7K	3	3	3	3	3	0	0
SM-4	0.7K	10	10	6	10	6	6	0
SM-5	1.7K	3	3	6	3	6	0	0
SM-6	0.4K	1	0	0	0	0	0	0
SM-7	1.1K	2	2	32	0	2	0	0
BIND-1	1.2K	1	1	35	1	33	0	0
BIND-2	1.7K	1	1	45	0	41	0	0
BIND-3	0.5K	1	1	4	0	1	0	0
BIND-4	1.1K	2	2	0	0	0	0	0
FTP-1	0.8K	4	4	13	4	3	0	0
FTP-2	1.5K	1	1	7	1	6	0	3
FTP-3	1.5K	24	24	25	23	17	7	12
polymorph-0.4.0	0.7K	10	10	6	3	6	0	6
ncompress-4.2.4	1.9K	12	0	10	4	0	0	0
129.compress	2.0K	7	7	34	7	14	4	7
spell-1.0	2.2K	1	0	0	0	0	0	0
man-1.5h1	4.7K	6	5	60	1	28	0	13
256.bzip2	4.7K	3	3	149	3	21	3	21
gzip-1.2.4a	8.2K	13	11	87	8	34	0	24
bc-1.06	17.0K	2	0	57	0	10	0	9
sed-4.0.8	25.9K	1	0	64	0	14	0	4
Total		138	118	677	100	264	33	104

Table 5.4: The number of alarms in taint analysis

Program	LOC	Bug	SOUND T	SOUND F	SELECTIVE T	SELECTIVE F	UNIFORM T	UNIFORM F
mp3rename-0.6	0.6K	1	1	0	1	0	1	0
ghostscript-8.71	1.5K	2	2	0	2	0	2	0
uni2ascii-4.14	5.7K	7	7	0	7	0	7	0
pal-0.4.3	7.4K	3	3	0	0	0	0	0
shntool-3.0.1	16.3K	1	1	10	1	1	1	0
sdop-0.61	23.9K	65	65	78	65	0	0	0
latex2rtf-2.3.8	28.7K	2	2	9	2	8	0	1
rrdtool-1.4.8	34.8K	1	1	12	1	1	1	0
daemon-0.6.4	58.4K	1	1	7	1	1	1	0
rplay-3.3.2	61.0K	3	3	7	2	4	1	2
urjtag-0.10	64.2K	12	12	78	6	0	0	0
a2ps-4.14	64.6K	6	6	26	3	12	1	0
dico-2.0	84.3K	2	2	46	1	1	1	2
Total		106	106	273	92	28	16	5

Chapter 6

Related Work

6.1 Parametric Static Analysis

In Chapter 3, our approach suggests novel techniques for *analysis-parameter inference* [26, 33, 38, 62]. There are many parameters to tune in static analysis, to improve either precision or scalability. The problem is how to find a set of minimal, or at least sufficient, parameters for the goal. Liang et al. [26] use machine learning to find a minimal context-sensitivity for given queries. Guided by the number of queries each analysis run has proven, the machine learning algorithm infers a minimal k value for each function. However, they study the minimal abstraction itself and provide no practical solutions for selective context-sensitivity. Zhang et al. [62] present a technique for finding the optimum abstraction, a cheapest abstraction that proves the query, but it is applicable only to disjunctive analyses. Naik et al. [33] use a dynamic analysis to select an appropriate parameter for a given query, while we use a static pre-analysis for parameter selection.

In Chapter 5, our work uses a parametric static analysis in a novel way, where the parameters specify the degree of soundness, not only the precision setting of the analysis. The existing parametric static analyses have been focused on balancing the precision and the cost of static analysis [33,37,38,62]. They infer a cost-effective abstraction for a newly given program by iterative refinements [33,62], impact pre-

analyses [37], or learning from a codebase [38]. On the other hand, our goal is to find a soundness parameter striking the right balance between existing fully sound and unsound approaches. Furthermore, the existing techniques for deriving static analysis parameters (e.g., [33,37,62]) cannot be used for our purpose since it is simply impossible to automatically judge truth and falsehood of alarms. We address this problem by designing a supervised learning method that learns a strategy from a given codebase with known bugs. Because we have labelled data, using the learning algorithm via black-box optimization [38] is inappropriate. Instead, we use an off-the-shelf learning method, which uses the gradient-based optimization algorithm and works much faster than the black-box optimization approach.

6.2 Goal-directed Static Analysis

While refinement-based analyses [15, 40, 55] are similar to our approach (in that they use a "pre-analysis" to adjust the main analysis precision), there is a fundamental difference in their techniques. Refinement-based approaches (e.g., client-driven analysis [15]) start with an *imprecise* analysis and refines the abstraction in response to client queries. On the other hand, our approach starts with a pre-analysis that estimates the impact of the *most precise* main analysis. As a result, our approach provides a precision guarantee, which does not hold in the refinement-based techniques. Furthermore, the principle behind our approach is general; it is applicable to a range of static analyses (such as interval and octagon analyses) with various precision axes (such as context-sensitivity and relational analysis). Existing refinement-based analyses have been special for pointer analyses [15, 40, 55].

Our approach is orthogonal to demand-driven analyses [17,55,56]. While demand-driven analyses aim to reduce analysis costs by computing only the partial solution necessary to answer given queries, we compute the exhaustive solution with an abstraction tailored to the queries (our analysis is run once for the entire set of queries). Both approach can complement each other.

6.3 Data-driven Static Analysis

Recently there have been a large amount of research activities for developing data-driven approaches to challenging program analysis problems, such as specification inference [42, 45], invariant generation [13, 34, 44, 49–51], acceleration of abstraction refinement [14], and smart report of analysis results [28,35,61]. In particular, Oh et al. [38] considered the problem of automatically learning analysis parameters from a codebase, which determine the heuristics used by the analysis. They formulated this parameter learning as a blackbox optimization problem, and proposed to use Bayesian optimization for solving the problem. Initially we followed this blackbox approach [38], and tried Bayesian optimization to learn a good variable-clustering strategy with our features. In the experiment, we learned the strategy from the small programs as in Section 4.5.2 and chose the top 200 variable pairs which are enough to make a good clustering as precise as our strategy; the learning process was too costly with larger training programs and more variable pairs. This initial attempt was a total failure. The learning process tried only 384 parameters and reduced 14 false alarms even during the learning phase for a whole week, while our strategy reduced 240 false alarms. Unlike the optimization problems for the analyses in [38], our problem was too difficult for Bayesian optimization to solve. We conjecture that this was due to the lack of smoothness in the objective function of our problem. This failure led to the approach in this paper, where we replaced the blackbox optimization by a much easier supervised-learning problem.

6.4 Context-sensitivity and Relational Analysis

Most of the previous context-sensitive analysis techniques assign contexts to calls in a uniform manner. The k-callstring approach (or k-CFA) [48,52] and its flexible variants [16], k-object sensitivity [29], and type sensitivity [4] are such cases. All these techniques generate calling contexts according to a single fixed policy and do not explore how to tune their parameters (for example, different k values at each call site) for target queries. The hybrid context-sensitivity [23], which employs mul-

tiple policies of assigning contexts in a single analysis, still does not tailor those policies to the program to analyze. There are also other approaches to context-sensitivity based on function summaries like [43], but here we do not discuss them as it is by itself a challenge to design a summary-based analysis with abstract domains of infinite height.

The scalability issue of the Octagon analysis is well-known, and there have been various attempts to optimize the analysis [36,53]. Oh et al. [36] exploited the data dependencies of a program and removed unnecessary propagation of information between program points during Octagon's fixpoint computation. Singh et al. [53] designed better algorithms for Octagon's core operators and implemented a new library for Octagon called OptOctagons, which has been incorporated in the Apron framework [20]. These approaches are orthogonal to our approach, and all of these three can be used together as in our implementation. We point out that although the techniques from these approaches [36, 53] improve the performance of Octagon significantly, without additionally making Octagon partial with good variable clusters, they were not enough to make Octagon scale large programs in our experiments. This is understandable because the techniques keep the precision of the original Octagon while making Octagon partial does not.

Existing variable-clustering strategies for the Octagon analysis use a simple syntactic criterion for clustering variables [2] (such as selecting variable pairs that appear in particular kinds of commands and forming one cluster for each syntactic block), or a pre-analysis that attempts to identify important variable pairs for Octagon [37]. When applied to large general-purpose programs (not designed for embedded systems), the syntactic criterion led to ineffective variable clusters, which made the subsequent partial Octagon analysis slow and fail to achieve the desired precision [37]. The approach based on the pre-analysis [37], on the other hand, has an issue with the cost of the pre-analysis itself; it is cheaper than that of Octagon, but it is still expensive as we showed in the paper. In a sense, our approach automatically learns fast approximation of the pre-analysis from the results of running the pre-analysis on programs in a given codebase. In our experiments, this ap-

proximation (which we called strategy) was 33x faster than the pre-analysis while decreasing the number of proved queries by 2% only.

6.5 Unsoundness in Static Analysis

Existing unsound static analyses are all uniformly unsound (e.g., [12, 22, 58–60]). In addition to their unsound handling of every loop and library call in a given program, they consider only a specific branch of all conditional statements in a program [22], deactivate all recursive calls [58, 59], or ignore all the possible inter-procedural aliasing [22, 58, 59]. As shown in this paper, these uniform approaches have a considerable drawback; it significantly impairs the capability of detecting real bugs. This paper is the first to tackle this problem and presents a novel approach of selectively employing unsoundness only when it is likely to be harmless.

Mangal et al. proposed an interactive system to control the unsoundness of static analysis online based on the user feedback [28]. They define a probabilistic Datalog analysis with "hard" and (unsound) "soft" rules, where the goal of the analysis is to find a solution that satisfies all of the hard rules while maximizing the weight of the satisfied soft rules. The feedback from analysis users is encoded as soft rules, and based on the feedback, the analysis is re-run and produces a report that optimizes the updated constraints. In our setting (non-Datalog), however, it is not straightforward to tune the unsoundness from user feedbacks. Instead, our approach automatically learns harmless unsoundness and selectively applies unsound strategies depending on the different circumstances.

Our goal is different from the existing work on unsoundness by Christakis et al. [6], which empirically evaluated the impact of unsoundness in a static analyzer using runtime checking. They instrumented programs with the unsound assumptions of the analyzer and check whether the assumptions are violated at runtime. On the contrary, we introduce a new notion of selective unsoundness and evaluate its impact in terms of the number of true alarms and false alarms reported.

Our approach is orthogonal to statistical post-processing of alarms [21, 24, 25]. The post-processing (e.g. ranking) approach aims to remove false positives (re-

ported false alarms). Instead, our approach aims to remove false negatives (unreported true alarms). From the undiscerning, uniformly unsound analysis that will have too many unreported true alarms, we tune it to be selectively unsound. These post-processing systems are also complementary to our approach. Because in practice any realistic bug-finding static analyzer cannot but be unsound (for the analysis precision and scalability), our technique provides a guide on how to design an unsound one. The existing post-processing techniques (e.g. ranking) can be anyway applied to the results from such selectively unsound static analyzers.

Chapter 7

Conclusion

We proposed selectively sensitive static analysis to balance among soundness, precision, and scalability of a static analyzer. The technical contribution of this dissertation is to design methods to select when and where the analysis turns on precise-yet-costly techniques.

- We proposed a general principle to design an impact pre-analysis. The pre-analysis estimates the impact of a precision-improving technique. Following this principle, we presented two program analyses that selectively apply precision-improving techniques, and demonstrated their effectiveness with experiments in a realistic setting. The first was a selective context-sensitive analysis that receives guidance from an impact pre-analysis. The experiments with realistic benchmarks showed that the method reduces a number of false alarms while only reasonably increasing the analysis cost.

- We proposed a method for learning a selection strategy for precision-improving techniques from the impact pre-analysis results on a codebase. Our method generates labeled data automatically from a given codebase by running the impact pre-analysis. The labeled data are then fed to an off-the-shelf classification algorithm, which infers a classifier that can identify important parts of a new unseen program, whose sensitivity should be turned on. Our experi-

ments show that this combination of pre-analysis and machine learning scales up well and is much faster than the one with only the impact pre-analysis.

- We proposed a method for learning a strategy for unsoundness heuristics. Given a codebase with known bugs and a static analyzer, we automatically generate labelled data and derive a classifier that captures the common characteristics of the harmless yet precision-decreasing program components. The experimental results showed that the technique is effectively applicable to two bug-finding static analyzers and reduces their false negative rates while retaining their original precision.

Bibliography

[1] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 196–207, 2003.

[3] Bernhard E. Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM, 1992.

[4] Martin Bravenboer, Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 17–30. ACM, 2011.

[5] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[6] Maria Christakis, Peter Müller, and Valentin Wüstholz. An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer. In *Pro-*

ceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation, pages 336–354, 2015.

[7] Common Vulnerabilities and Exposures. https://cve.mitre.org.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[9] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[10] Alain Deutsch. On the complexity of escape analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 358–371. ACM, 1997.

[11] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–12. ACM, 2012.

[12] Cormac Flanagan, K . Rustan M . Leino, Mark Lillibridge, Greg Nelson, and James B . Saxe. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 1–1. Springer Berlin Heidelberg, 2004.

[13] Pranav Garg, Daniel Neider, P Madhusudan, Dan Roth, Pranav Garg, Daniel Neider, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 499–512. ACM, 2016.

[14] Radu Grigore, Radu Grigore, and Hongseok Yang. Abstraction refinement guided by a learnt probabilistic model. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 485–498. ACM, 2016.

[15] Samuel Z. Guyer and Calvin Lin. Client-Driven Pointer Analysis. In *Proceedings of the 10th International Conference on Static Analysis*, pages 214–236. Springer Berlin Heidelberg, 2003.

[16] Williams Ludwell Harrison. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3-4):179–396, 1989.

[17] Nevin Heintze, Olivier Tardieu, and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 24–34. ACM, 2001.

[18] Kihong Heo, Hakjoo Oh, and Hongseok Yang. Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 237–256. Springer, Berlin, Heidelberg, 2016.

[19] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-Learning-Guided Selectively Unsound Static Analysis. In *Proceedings of the 39th International Conference on Software Engineering*, 2017.

[20] Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 661–667. Springer Berlin Heidelberg, 2009.

[21] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming False Alarms from a Domain-Unaware C Analyzer by a Bayesian Statistical Post

Analysis. In *Proceedings of the 10th International Conference on Static Analysis*, pages 203–217. Springer Berlin Heidelberg, 2005.

[22] Yungbum Jung and Kwangkeun Yi. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th international symposium*, pages 131–140. ACM, 2008.

[23] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 423–434. ACM, 2013.

[24] Ted Kremenek, Ken Ashcraft, Junfeng Yang, Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 83–93. ACM, 2004.

[25] Ted Kremenek and Dawson Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Proceedings of the 10th International Conference on Static Analysis*, pages 295–315. Springer Berlin Heidelberg, 2003.

[26] Percy Liang, Omer Tripp, Percy Liang, Omer Tripp, and Mayur Naik. Learning minimal abstractions. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 31–42. ACM, 2011.

[27] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, pages 1–5, 2005.

[28] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 462–473. ACM, 2015.

[29] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11. ACM, 2002.

[30] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[31] Tom M. Mitchell. *Machine learning*. McGraw-Hill, Inc., 1997.

[32] Kevin P. Murphy. *Machine learning: a probabilistic perspective (adaptive computation and machine learning series)*. Mit Press ISBN, 2012.

[33] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. Abstractions from tests. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 2012.

[34] Aditya V. Nori and Rahul Sharma. Termination proofs from tests. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 246–256. ACM, 2013.

[35] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, Yves Le Traon, Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 469–484. ACM, 2016.

[36] Hakjoo Oh, Kihong Heo, Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 229–238. ACM, 2012.

[37] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 475–484. ACM, 2014.

[38] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 572–588. ACM Press, 2015.

[39] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[40] John Plevyak, Andrew A Chien, John Plevyak, and Andrew A Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 324–340. ACM, 1994.

[41] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439. ACM, 2016.

[42] Veselin Raychev, Pavol Bielik, Andreas Krause, Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 761–774. ACM, 2016.

[43] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 49–61. ACM, 1995.

[44] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 295–306. ACM, 2008.

[45] Sriram Sankaranarayanan, Franjo Ivančić, and Aarti Gupta. Mining library specifications using inductive logic programming. In *Proceedings of the 30th International Conference on Software Engineering*, pages 131–140. ACM, 2008.

[46] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001.

[47] Scikit-learn. `http://scikit-learn.org`.

[48] Micha Sharir and Amir Pnueli. Two Approach to Interprocedural Data Flow Analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice-Hall, 1981.

[49] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A Data Driven Approach for Algebraic Loop Invariants. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, pages 574–592. Springer Berlin Heidelberg, 2013.

[50] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. Verification as Learning Geometric Concepts. In *Proceedings of the 20th International Conference on Static Analysis*, pages 388–411. Springer Berlin Heidelberg, 2013.

[51] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as Classifiers. In *Proceedings of the th International Conference on Computer Aided Verification*, pages 71–87. Springer Berlin Heidelberg, 2012.

[52] Olin Shivers. *Control-flow analysis of higher-order languages or Taiming Lambda*. PhD thesis, CMU, 1991.

[53] Gagandeep Singh, Markus Püschel, and Martin Vechev. Making Numerical Program Analysis Fast. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015.

[54] Sparrow. `http//ropas.snu.ac.kr/sparrow`.

[55] Manu Sridharan, Rastislav Bodík, and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400. ACM, 2006.

[56] Manu Sridharan, Denis Gopan, Lexin Shan, Rastislav Bodík, and Lexin Shan. Demand-driven points-to analysis for Java. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 59–76. ACM, 2005.

[57] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM, 2004.

[58] Yichen Xie and Alex Aiken. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 115–125. ACM, 2005.

[59] Yichen Xie and Alex Aiken. Saturn: a SAT-based tool for bug detection. In *Proceedings of the 17th International Conference on Computer Aided Verification*. Springer-Verlag, 2005.

[60] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 2002*

ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 327–336. ACM, 2003.

[61] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static C analyzer. *Information Processing Letters*, 102(2-3):118–123, 2007.

[62] Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Language, and Applications*, pages 365–376. ACM, 2013.

[63] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering.* ACM, 2004.

초 록

이 학위 논문에서는 정적 분석 성능을 결정짓는 세 가지 축인 안전성 (soundness), 정확도 (precision), 확장성 (scalability) 을 최대한 달성할 수 있는 방법을 제시한다. 정적 분석에는 여러가지 정확도 상승 기법들이 있지만, 무턱대고 적용할 시에는 분석이 심각하게 느려지거나 실제 실행 의미를 지나치게 많이 놓치는 문제가 있다. 이 논문의 핵심은, 이렇게 정확하지만 비용이 큰 분석 기법이 꼭 필요한 곳만을 선별해내는 기술이다. 먼저, 정확도 상승 기법이 꼭 필요한 부분을 예측하는 또 다른 정적 분석인 예비 분석을 제시한다. 본 분석은 이 예비 분석의 결과를 바탕으로 정확도 상승 기법을 선별적으로 적용함으로서 효율적으로 분석을 할 수 있다. 또한, 기계학습을 이용하여 과거 분석 결과를 학습함으로써 더욱 효율적으로 선별할수 있는 기법을 제시한다. 학습에 쓰이는 데이터는 앞서 제시한 예비 분석과 본 분석을 여러 학습 프로그램에 미리 적용한 결과로부터 자동으로 얻어 낸다. 여기서 제시한 방법들은 실제 C 소스 코드 분석기에 적용하여 그 효과를 실험적으로 입증했다.

주요어 : 프로그래밍 언어, 정적 분석, 선별, 예비 분석, 기계 학습
학 번 : 2009-20919