A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis

MINSEOK JEON, SEHUN JEONG[*], SUNGDEOK CHA, and HAKJOO OH[†], Korea University, Republic of Korea

We present a new machine-learning algorithm with disjunctive model for data-driven program analysis. One major challenge in static program analysis is a substantial amount of manual effort required for tuning the analysis performance. Recently, data-driven program analysis has emerged to address this challenge by automatically adjusting the analysis based on data through a learning algorithm. Although this new approach has proven promising for various program analysis tasks, its effectiveness has been limited due to simple-minded learning models and algorithms that are unable to capture sophisticated, in particular disjunctive, program properties. To overcome this shortcoming, this article presents a new disjunctive model for data-driven program analysis as well as a learning algorithm to find the model parameters. Our model uses boolean formulas over atomic features and therefore is able to express nonlinear combinations of program properties. Key technical challenge is efficiently determine a set of good boolean formulas as brute-force search would simply be impractical. We present a stepwise and greedy algorithm that efficiently learns boolean formulas. We show the effectiveness and generality of our algorithm with two static analyzers: context-sensitive points-to analysis for Java and flow-sensitive interval analysis for C. Experimental results show that our automated technique significantly improves the performance of the state-of-the-art techniques including ones hand-crafted by human experts.

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

Additional Key Words and Phrases: Data-driven program analysis, Static analysis, Context-sensitivity, Flow-sensitivity

ACM Reference Format:
Minseok Jeon, Sehun Jeong, Sungdeok Cha, and Hakjoo Oh. 2017. A Machine-Learning Algorithm with Disjunctive Model for Data-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 9, 4, Article 39 (December 2017), 42 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

One major challenge in static program analysis is a substantial amount of manual effort required for tuning the analysis performance for real-world applications. Practical static analysis tools use a variety of heuristics to optimize their performance. For example, context-sensitivity is essential for analyzing object-oriented programs, as it distinguishes method's local variables and objects in different calling-contexts. However, applying context-sensitivity to all methods in the program does not scale and therefore real-world static analyzers apply context-sensitivity only to profitable methods determined by some heuristic rules [Smaragdakis et al. 2014]. Another example is a relational analysis such as ones with Octagons [Miné 2006]. Because it is impractical to keep track of all variable relationships in the program, static analyzers employ variable-clustering heuristics

[*] The first and second authors contributed equally to this work
[†] Corresponding author

Authors' address: Minseok Jeon, minseok_jeon@korea.ac.kr; Sehun Jeong, gifaranga@korea.ac.kr; Sungdeok Cha, scha@korea.ac.kr; Hakjoo Oh, hakjoo_oh@korea.ac.kr, Department of Computer Science and Engineering, Korea University, 145, Anam-ro, Sungbuk-gu, Seoul, 02841, Republic of Korea.

2009. 0164-0925/2017/12-ART39 $15.00
https://doi.org/0000001.0000001

to apply Octagons only to small sets of program variables [Blanchet et al. 2003; Miné 2006]. Other examples include heuristics for trace partitioning (e.g. when to split and merge traces) [Rival and Mauborgne 2007] and heuristics for clumping abstract states in disjunctive analysis [Li et al. 2017], to name a few. In practice, the qualities of these heuristics determine the final benefit of static analysis tools. However, manually-designing such heuristics is nontrivial and requires a huge amount of laborious work and expert knowledge.

Recently *data-driven program analysis* has emerged to address this challenge of manually tuning program analysis tools [Cha et al. 2016; Chae et al. 2017; Heo et al. 2016, 2017b; Jeong et al. 2017; Oh et al. 2015]. In this approach, program-analysis heuristics are automatically generated from codebases through learning algorithms with little reliance on analysis designers. Instead of manually developing a fixed heuristic from scratch, it employs parametric heuristic rules and learning algorithms to find the parameter values that maximize the analysis performance over the codebases. This data-driven approach has proven effective at generating various analysis heuristics automatically: e.g., heuristics for applying flow-sensitivity [Oh et al. 2015], variable clustering [Heo et al. 2016], widening thresholds [Cha et al. 2016], and unsoundness [Heo et al. 2017b].

However, current techniques for data-driven program analysis are limited to rather simple machine-learning models and algorithms, which are inherently short of capturing sophisticated program properties that are especially hard to find manually. For example, Oh et al. [2015] proposed a machine-learning model and algorithm based on the linear combination of feature vectors and numeric parameters, akin to a linear regression in typical machine learning applications. While the method is simple and generalizing well to unseen programs, the algorithm based on the linear model cannot capture nontrivial, in particular disjunctive, program features, which is often needed to generate high-quality analysis heuristics. For example, as we demonstrate in this article, learning a cost-effective heuristic for context-sensitive points-to analysis for Java is beyond the capability of the previous linear algorithms.

In this article, we present a novel learning algorithm that overcomes the existing limitation. Our algorithm is based on a parameterized heuristic that is able to express disjunctive properties of program elements. Our algorithm is generally applicable to any program analysis based on k-limited abstractions (e.g. k-CFA). Let us describe our algorithm with k-CFA-based context-sensitive analysis. When k (i.e. the maximum context depth to maintain) is given, we use a set of k boolean formulas: $\{f_1, f_2, \ldots, f_k\}$ where f_i is a boolean combination of the atomic features that captures complex and high-level properties of a method. Each atomic feature describes a low-level property such as whether a method has an allocation statement or not. Context-sensitivity of depth i is applied only to the methods whose properties are described by the formula f_i. This way, the analysis applies context-sensitivity selectively and the selection is determined by the k boolean formulas. Key technical challenge in this approach is to efficiently determine a good set of boolean formulas as brute-force search would simply be impractical. We demonstrate that it is possible to reduce the problem of simultaneously learning k boolean formulas into a set of k sub-problems of finding each formula, drastically reducing the search space. We use a greedy algorithm to solve each sub-problem, which produces accurate yet general formulas by iteratively refining the formulas while keeping them in disjunctive normal form.

Experimental results show that our learning algorithm produces highly cost-effective program-analysis heuristics for two instance analyses: context-sensitive points-to analysis for Java and flow-sensitive interval analysis for C. On top of the Doop framework [Bravenboer and Smaragdakis 2009], we applied our algorithm to three context-sensitive analyses: selective object-sensitivity [Kastrinis and Smaragdakis 2013b], object-sensitivity [Milanova et al. 2005], and type-sensitivity [Smaragdakis et al. 2011]. In all analyses, the results show that our approach strikes a good balance between precision and scalability trade-offs. For instance, when we applied our technique to selective 2-object-sensitivity (*S2objH*), the resulting analysis has virtually the same scalability of the context-insensitive analysis while enjoying most of the precision benefits of the most precise analysis. In particular, our data-driven points-to analysis excels the performance of the existing manually-crafted state-of-the-art, namely introspective analyses [Smaragdakis et al. 2014], in terms of precision and speed while the prior algorithm based on linear model by Oh et al. [2015] only generated heuristics that lag behind the manually-written heuristics. We also implemented our approach in Sparrow [Oh et al. 2012] and confirmed that it generates heuristics that outperform for controlling flow-sensitivity of interval analysis for C.

Contributions. In summary, our key contributions are as follows:

- We present a new learning algorithm for data-driven program analysis. Although the idea of data-driven program analysis itself is not new [Cha et al. 2016; Heo et al. 2016, 2017b; Oh et al. 2015], we make two novel contributions: use of nonlinear model for program-analysis heuristics (Section 4.1) and efficient learning algorithm (Section 4). On the other hand, existing data-driven program analyses rely on simple linear models [Cha et al. 2016; Oh et al. 2015] or off-the-shelf models in the presence of labelled data [Heo et al. 2016, 2017b].
- We demonstrate the effectiveness of our approach with two instance analyses: context-sensitive points-to analysis for Java and flow-sensitive interval analysis for C. We also demonstrate that use of nonlinear model is a key to success; without it, the analysis becomes significantly less precise and costly (Section 7.2).

Comparison with Previous Version. This article supersedes its previous version [Jeong et al. 2017] presented at *ACM Conference on Object-Oriented Programming, Languages, Systems, and Applications 2017 (OOPSLA 2017)*. In comparison, this article makes the following extensions:

- It presents our learning algorithm in a general setting (Sections 3, 4.1, 4). The previous paper has focused on context-sensitivity for points-to analysis and the algorithm was tightly coupled with the analysis. We generalized the presentation so that the algorithm is applicable to any static analyses with k-limited abstractions.
- It shows the generality of the algorithm with a new instance analysis. We describe how to apply the algorithm to a flow-sensitive interval analysis for C (Section 6) and provide experimental evaluation with the analysis (Section 7.4).
- It provides an overview section with a running example to give a high-level ideas underlying our approach and learning algorithm (Section 2).
- It elaborates on the instance analysis of points-to analysis for Java (Section 5). The previous paper has omitted descriptions on the details of Datalog-based points-to analysis for Java.

2 OVERVIEW

We illustrate our approach using a context-sensitive points-to analysis for Java.

2.1 Selective Context-Sensitivity

Suppose we analyze the example program in Fig. 1 with a flow-insensitive and context-sensitive subset-based points-to analysis. In this section, we consider call-site-sensitivity (k-CFA) and fix k to 2. However, our approach is not limited to this particular analysis; for example, it also applies to object-sensitivity and type-sensitivity with any k. The example program consists of five classes: `A`, `B`, `C`, `D`, and `E`. The class `C` has three methods (`dummy`, `id1`, and `id2`). Methods `id1` and `dummy` are called from `B.m`. In `main`, `B.m` is called twice. With the analysis, we would like to prove that the two type-casting operations at lines 10 and 11 are safe; they perform safe down-casting at runtime.

```
1  class D {} class E {}
2  class C {
3    void dummy(){}
4    Object id1(Object v){ return id2(v); }
5    Object id2(Object v){ return v; }
6  }
7  class B {
8    void m (){
9      C c = new C();
10     D d = (D)c.id1(new D());   //Query 1
11     E e = (E)c.id1(new E());   //Query 2
12     c.dummy();
13   }
14 }
15 public class A {
16   public static void main(String[] args){
17     B b = new B();
18     b.m();
19     b.m();
20   }
21 }
```

Fig. 1. Example program

Note that context-insensitive points-to analysis cannot prove any of the queries. Because it is insensitive to its calling contexts, the objects from allocation-sites at lines 10 and 11 get conflated and therefore the argument (`v`) of method `id1` points to objects of classes `D` and `E`. Eventually, these two types of objects get propagated back to the results of the method calls at lines 10 and 11, making the analysis concludes that the subsequent type-casts are potentially unsafe.

On the other hand, the conventional 2-call-site-sensitive analysis is able to prove the safety of the queries as the methods `id1` and `id2` are analyzed separately for their calling-contexts. For example, method `id2` is analyzed separately for the calling contexts $4 \cdot 10$ and $4 \cdot 11$, where $a \cdot b$ denotes a sequence of call-sites a and b (a is the most recent call-site) and we use line numbers as call-sites. However, the 2-call-site-sensitive points-to analysis is typically too expensive to scale up to large Java programs.

To achieve both precision and scalability, practical static analysis tools often apply context-sensitivity selectively. For instance, to prove the queries in the example program, it is sufficient to use 2-call-site-sensitivity only for `C.id2`, 1-call-site-sensitivity for `C.id1`, and context-insensitivity for other methods (`B.m` and `C.dummy`). That is, the analysis uses the following information

$$\{\texttt{C.id2} \mapsto 2, \quad \texttt{C.id1} \mapsto 1, \quad \texttt{B.m} \mapsto 0, \quad \texttt{C.dummy} \mapsto 0\}$$

that maps methods to their appropriate context-depths ($k = 0$, 1, or 2). The main challenge in this approach is to determine the amount of context-sensitivity required for each method. Typically this kind of decisions has been made by hand-crafted heuristics, e.g., [Smaragdakis et al. 2014]. Our goal is to automate this process and generate such heuristics from data through a learning algorithm.

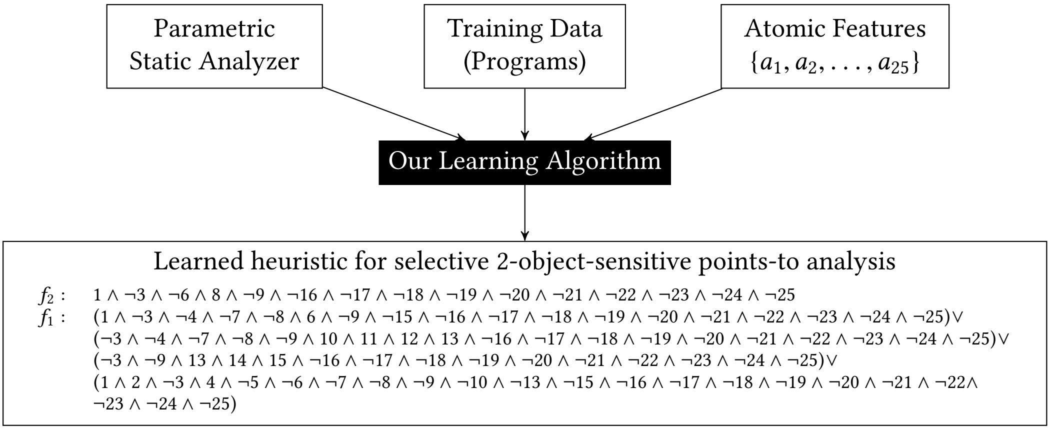
Parametric Static Analyzer
Training Data (Programs)
Atomic Features $\{a_1, a_2, \ldots, a_{25}\}$
Our Learning Algorithm
Learned heuristic for selective 2-object-sensitive points-to analysis
f_2 : $1 \land \neg 3 \land \neg 6 \land 8 \land \neg 9 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25$
f_1 : $(1 \land \neg 3 \land \neg 4 \land \neg 7 \land \neg 8 \land 6 \land \neg 9 \land \neg 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25) \lor$
$(\neg 3 \land \neg 4 \land \neg 7 \land \neg 8 \land \neg 9 \land 10 \land 11 \land 12 \land 13 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25) \lor$
$(\neg 3 \land \neg 9 \land 13 \land 14 \land 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25) \lor$
$(1 \land 2 \land \neg 3 \land 4 \land \neg 5 \land \neg 6 \land \neg 7 \land \neg 8 \land \neg 9 \land \neg 10 \land \neg 13 \land \neg 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land$
$\neg 23 \land \neg 24 \land \neg 25)$

Fig. 2. Overview of our approach. Our algorithm takes static analyzer, training programs, and atomic features. When we learn context-sensitivity heuristics for points-to analysis, we used 25 atomic features of methods. As output, our algorithm produces boolean formulas over the atomic features. f_1 and f_2 above show the formulas that are actually found by our learning algorithm for selective 2-object-sensitivity (*S2objH*).

2.2 Approach Overview

Fig. 2 shows input and output of our algorithm. As input, our learning algorithm takes three components: parametric static analyzer, training data, and atomic features. The static analyzer is a parametric one that is able to assign different context-depths to each method, the training data is just a collection of programs (without any labeled data), and atomic features are predicates on methods that describe syntactic/semantic properties of methods, e.g., methods having invocation statement in their body, methods that return strings, etc. For example, suppose we have five atomic features:

$$\mathbb{A} = \{a_1, a_2, a_3, a_4, a_5\}.$$

and assume that the methods in Fig. 1 have the features as follows:

$$\begin{array}{rcl} \texttt{C.id2} & : & \{a_1, a_3, a_5\} \\ \texttt{C.id1} & : & \{a_3, a_5\} \\ \texttt{B.m} & : & \{a_1, a_2, a_3\} \\ \texttt{B.dummy} & : & \{a_2, a_3, a_4\} \end{array} \tag{1}$$

The method `C.id2` has features a_1, a_3, a_5, method `C.id1` has a_3, a_5, and so on.

Given the three components as input, our learning algorithm produces as output two boolean formulas f_1 and f_2 over the atomic features, e.g.,

$$f_1 = \neg a_4 \land a_5, \qquad f_2 = (a_1 \land a_5) \lor (a_2 \land \neg a_3).$$

The formulas f_1 and f_2 describe the methods that are determined by our algorithm to receive 1-CFA and 2-CFA, respectively. In this case, the formula f_1 describes the set of methods that have the feature a_5 but not a_4. The formula f_2 denotes the set of methods that have the conjunctive features $a_1 \land a_5$ or $a_2 \land \neg a_3$. Therefore, with the method features in (1), the formulas f_1 and f_2 represent the following sets of methods:

$$f_1 : \{\texttt{C.id1}, \texttt{C.id2}\}, \qquad f_2 : \{\texttt{C.id2}\}$$

The learned formulas f_1 and f_2 are used for analyzing new unseen programs. For example, when analyzing the program in Fig. 1, our analysis performs a selectively context-sensitive analysis by applying 2-call-site-sensitivity to `C.id2`, 1-call-site-sensitivity to `C.id1`, and context-insensitivity to the remaining methods. Note that we assign the greater context-depth 2 to `C.id2` when it belongs to both f_1 and f_2.

Fig. 2 shows the formulas f_1 and f_2 that are actually found by our algorithm for selective 2-object-sensitive points-to analysis for Java. For example, the learned formula f_2 describes the set of methods that have the atomic feature a_1, does not have a_3, does not have a_6, does have a_8, and so on. The formula f_1 is a disjunction of four different conjunctive formulas. In practice the heuristic with these formulas performs extremely well compared to the existing heuristics manually-tuned by analysis experts (Section 7). Note that the learned formulas are too complex to be discovered manually. The main strength of our approach is to find such a nontrivial heuristic automatically from data.

2.3 Our Learning Algorithm

Now we illustrate how our algorithm finds the formulas from an existing codebase. For simplicity, we assume that the codebase consists of the single program in Fig. 1. In practice, we use exclusive sets of programs for training and testing. We also assume that the maximum context-depth is 2 (i.e. $k = 2$). Thus, our goal is to learn formulas f_1 and f_2.

Informally, the learning problem is formulated as the following optimization problem:

Find f_1 and f_2 that minimize the analysis cost while maintaining the analysis precision.

That is, we would like to find boolean formulas f_1 and f_2 such that the selective context-sensitive analysis with f_1 and f_2 satisfies some precision constraint while minimizing the analysis cost. The desired constraint on precision is given by users. In this section, we consider the precision constraint that prescribes the analysis to prove both queries in the example program, i.e., having the same precision as 2-CFA.

Our algorithm learns f_1 and f_2 in a stepwise fashion. We begin with learning f_2 while fixing f_1 to *true*; that is, we apply 2-CFA to the methods described by f_2 and 1-CFA to all the remaining methods (i.e. methods that satisfy the formula *true*). The intuition is that we first identify the methods whose precision improves with 2-CFA but not with 1-CFA. After that, we continue to learn the methods that require 1-CFA to improve the precision. Below, we illustrate how to learn f_2. Learning f_1 can be done with the same procedure.

Our algorithm learns f_2 by iteratively running the analysis on the training program at different precision levels. Initially, it analyzes the program with full precision; that is, the algorithm maintains the formula in disjunctive normal form (DNF) and initially sets f_2 to the most general formula:

$$f_2 = a_1 \vee \neg a_1 \vee a_2 \vee \neg a_2 \vee \cdots \vee a_5 \vee \neg a_5$$

Note that this formula is equivalent to *true* (i.e. denoting all methods in the program) and therefore the analysis satisfies the given precision constraint (i.e. proving both queries). The algorithm iteratively strengthens each (conjunctive) clause until it finds a formula with a minimal cost while still proving the queries. At any iteration, the formula is a disjunction of conjunctions of literals:

$$f_2 = c_1 \vee c_2 \vee \cdots \vee c_m$$

where c_i is a conjunction of literals (i.e. atomic features or their negations). One iteration of the algorithm consists of the following steps:

(1) Choose the most expensive clause c_i from f_2 (i.e., the clause c_i that makes the analysis slowest). The individual cost of each conjunction c_i can be obtained by analyzing the example

program with $f_1 = \textit{true}$ and $f_2 = c_i$. Thus, we choose the clause c_i such that the analysis with $f_2 = c_i$ and $f_1 = \textit{true}$ is more expensive than those with $f_2 = c_j$ and $f_1 = \textit{true}$ for all $j \neq i$. Here, our algorithm is greedy and picks the most expensive clause to reduce the analysis cost as much as possible in a single step of refinement.

(2) Strengthen the chosen clause. We conjoin the chosen clause c_i with an atomic feature $a_k \in \mathbb{A}$ (or $\neg a_k$) that does not appear in c_i:

$$f_2' = c_1 \vee c_2 \vee \cdots (c_i \wedge a_k) \cdots \vee c_m.$$

This refinement is likely to reduce analysis cost but it decreases precision as well. In order not to lose too much precision, we conservatively choose a_k such that the resulting formula decreases the analysis precision as least as possible. That is, among the formulas

$$\begin{array}{rcll} f'_{2,a_1} & = & c_1 \vee c_2 \vee \cdots (c_i \wedge a_1) \cdots \vee c_m & \\ f'_{2,\neg a_1} & = & c_1 \vee c_2 \vee \cdots (c_i \wedge \neg a_1) \cdots \vee c_m & \\ & \vdots & & \\ f'_{2,a_k} & = & c_1 \vee c_2 \vee \cdots (c_i \wedge a_k) \cdots \vee c_m & (a_k \notin c_i) \\ f'_{2,\neg a_k} & = & c_1 \vee c_2 \vee \cdots (c_i \wedge \neg a_k) \cdots \vee c_m & (\neg a_k \notin c_i) \\ & \vdots & & \\ f'_{2,a_n} & = & c_1 \vee c_2 \vee \cdots (c_i \wedge a_n) \cdots \vee c_m & \\ f'_{2,\neg a_n} & = & c_1 \vee c_2 \vee \cdots (c_i \wedge \neg a_n) \cdots \vee c_m & \end{array}$$

we choose one that leads to the most precise analysis.

(3) Check whether the formula f_2' still satisfies the desired precision (e.g. proving both queries).
- If f_2' satisfies the condition, we further check whether the refined clause $c_i \wedge a_k$ is semantically subsumed by other clauses in f_2'. If so, we drop the clause $c_i \wedge a_k$ from f_2'. Next, we go back to step (2) and continue to strengthen f_2' further.
- If f_2' does not satisfy the precision condition, we rollback the latest change to f_2' and go back to step (1) to pick another clause (other than c_i) to refine.

(4) The above procedure terminates when no longer clauses remain to refine. The formula f_2 on termination is returned.

For example, suppose at some iteration the formula f_2 is given as follows:

$$f_2 = a_1 \vee (a_2 \wedge \neg a_3)$$

At this point, we choose the most expensive conjunction in f_2. Suppose that the analysis with $f_2 = a_1$ is more expensive than that with $f_2 = a_2 \wedge \neg a_3$. Then, our algorithm chooses the clause a_1 and attempts to refine it.

To refine a_1, we choose an atom $a_k \in \mathbb{A}$ such that $a_k \neq a_1$, the analysis with $f_2 = a_1 \wedge a_k$ satisfies the precision constraint (i.e. proving the two queries), and the precision loss due to the new conjunct gets minimized. If no such atomic feature exists, we do not refine a_1 and move on to the next conjunction. Suppose a_5 is the atom that satisfies the three conditions. Then, the formula f_2 gets strengthened as follows:

$$f_2' = (a_1 \wedge a_5) \vee (a_2 \wedge \neg a_3)$$

Now the algorithm goes for another round of refinement of f_2'. The algorithm terminates when every conjunction in f_2' fails to be refined.

3 SETTING

In this section, we define the class of static program analyses to which our approach is applicable.

3.1 Parametric Program Analysis

Let $P \in \mathbb{P}$ be a program to analyze. Let $\mathbb{J}_P$ be an index set of program components that represent parts of P. For example, in a partially context-sensitive analysis, $\mathbb{J}_P$ may represent the set of procedures, methods, or call-sites in the program. In a partially flow-sensitive analysis, $\mathbb{J}_P$ may represent the set of program variables. Let k be a fixed non-negative integer that indicates the degree of abstraction. Then, we define the set $\mathcal{A}_P$ of abstractions for P as follows:

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

Abstractions are vectors of natural numbers in $\{0, 1, \dots, k\}$ with indices in $\mathbb{J}_P$, and are ordered pointwise:

$$\mathbf{a} \sqsubseteq \mathbf{a}' \iff \forall m \in \mathbb{J}_P.\ \mathbf{a}_m \le \mathbf{a}'_m.$$

Intuitively, $\mathbf{a}_m = i$ means that we analyze the program part m at the level of the abstraction i. For example, in a partially context-sensitive analysis, this means that the procedure m is analyzed with context-sensitivity of depth i (i.e. the analysis distinguishing the last i context elements of the procedure).

Note that this family of abstractions is general enough to cover commonly-used abstractions in program analysis. For example, the parametric frameworks for k-call-site/object sensitivity [Milanova et al. 2005] and numerical analysis [Cha et al. 2016; Oh et al. 2015] belong to this family.

We can regard an abstraction $\mathbf{a} \in \mathcal{A}_P$ as a function from $\mathbb{J}_P$ to $\{0, 1, \dots, k\}$:

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

We write $\mathbf{k}$ and $\mathbf{0}$ for the most precise and least precise abstractions, respectively:

$$\mathbf{k} = \lambda m \in \mathbb{J}_P.\ k, \qquad \mathbf{0} = \lambda m \in \mathbb{J}_P.\ 0$$

For instance, in a partially context-sensitive analysis, the analysis with $\mathbf{k}$ represents the standard k-limiting analysis while $\mathbf{0}$ means the context-insensitive analysis.

We assume that a set $\mathbb{Q}_P$ of assertions is given together with P. For instance, $\mathbb{Q}_P$ is the set of all type casts in P and the analysis attempts to prove that they do not fail at runtime. We model program analysis for P by the function:

$$F_P : \mathcal{A}_P \to \wp(\mathbb{Q}_P) \times \mathbb{N}.$$

Given a program P, the analysis takes an abstraction $\mathbf{a} \in \mathcal{A}_P$ of the program and returns a pair (Q, n) of the set $Q \subseteq \mathbb{Q}_P$ of assertions proved by the analysis and the natural number $n \in \mathbb{N}$ that represents the cost (e.g., time) of the analysis with the abstraction $\mathbf{a}$. For instance, Q denotes the set of type casts proved to be safe by the analysis. We define two projection functions: $\mathsf{proved}(F_P(\mathbf{a}))$ and $\mathsf{cost}(F_P(\mathbf{a}))$ denote the set of proved assertions (Q) and the cost (n) of the analysis $F_P(\mathbf{a})$, respectively.

In this section, we assume that the analysis is monotone in the following sense:

Definition 3.1 (Monotonicity of Analysis). Let $P \in \mathbb{P}$ be a program and $\mathbf{a}, \mathbf{a}' \in \mathcal{A}_P$ be abstractions of P. We say the analysis F_P is monotone if the following condition holds:

$$\mathbf{a} \sqsubseteq \mathbf{a}' \implies \mathsf{proved}(F_P(\mathbf{a})) \subseteq \mathsf{proved}(F_P(\mathbf{a}')).$$

That is, we assume that more precise abstractions lead to proving more assertions. This condition is required for our learning algorithm to guarantee precision. In this paper, we also generally assume that the analysis cost monotonically increases with respect to the order of the abstraction. Our algorithm does not strictly enforce this property (which is not always true in static analysis [Oh 2009]) but it is designed with this property in mind. Therefore, our approach may not be directly applicable to static analysis of JavaScript programs, where the analysis with the least precise abstraction is usually not the cheapest one.

3.2 Goal

Suppose we have a codebase $\mathbf{P} = \{P_1, P_2, \ldots, P_m\}$, which is a collection of programs. Our goal is to automatically learn from $\mathbf{P}$ a *abstraction-selection heuristic* $\mathcal{H}$:

$$\mathcal{H}(P) : \mathbb{J}_P \to \{0, 1, \ldots, k\}$$

which takes a program P and returns an abstraction (i.e., an assignment of abstraction levels to each program component) of the program. Once $\mathcal{H}$ is learned from the codebase, it is used to analyze previously unseen program P as follows:

$$F_P(\mathcal{H}(P)).$$

Our aim is to learn from $\mathbf{P}$ a good heuristic $\mathcal{H}$ such that the precision of the analysis $F_P(\mathcal{H}(P))$ is close to that of the most precise analysis $F_P(\mathbf{k})$ while its cost is comparable to that of the least precise analysis $F_P(\mathbf{0})$.

4 OUR LEARNING ALGORITHM

In this section, we present our learning algorithm. We present our parameterized heuristic based on boolean formulas (Section 4.1), define the learning objective (Section 4.2), and present our learning algorithm (Section 4.3).

4.1 A Disjunctive Parameterized Heuristic

To enable learning, we first need to define a hypothesis space of the heuristics, which is called model or inductive bias in the machine learning community. That is, we need to choose and represent a model which is a restricted subset of the entire selection heuristics. We use a nonlinear, disjunctive model that combines atomic features with boolean formulas.

We assume that a set of *atomic features* is given: $\mathbb{A} = \{a_1, a_2, \ldots, a_n\}$. An atomic feature a_i describes a property of program components; it is a function from programs to predicates on program components:

$$a_i(P) : \mathbb{J}_P \to \{\mathit{true}, \mathit{false}\}.$$

Next, we define the following set of boolean formulas over the atomic features:

$$f \to \mathit{true} \mid \mathit{false} \mid a_i \in \mathbb{A} \mid \neg f \mid f_1 \wedge f_2 \mid f_1 \vee f_2$$

Given a program P, a boolean formula f means a set of program components, denoted $[\![f]\!]_P$:

$$\begin{array}{rlrl} [\![\mathit{true}]\!]_P &= \mathbb{J}_P & [\![\neg f]\!]_P &= \mathbb{J}_P \setminus [\![f]\!]_P \\ [\![\mathit{false}]\!]_P &= \emptyset & [\![f_1 \wedge f_2]\!]_P &= [\![f_1]\!]_P \cap [\![f_2]\!]_P \\ [\![a_i]\!]_P &= \{m \in \mathbb{J}_P \mid a_i(P)(m) = \mathit{true}\} & [\![f_1 \vee f_2]\!]_P &= [\![f_1]\!]_P \cup [\![f_2]\!]_P \end{array}$$

Suppose we are given a vector Π of k boolean formulas:

$$\Pi = \langle f_1, \ldots, f_k \rangle.$$

This vector will become the parameter of our model. Given a parameter $\Pi = \langle f_1, \ldots, f_k \rangle$, we define the model (i.e., parameterized heuristic), denoted $\mathcal{H}_\Pi$, as follows:

$$\mathcal{H}_\Pi(P) = \lambda m \in \mathbb{J}_P. \begin{cases} k & \text{if } m \in [\![f_k]\!]_P \\ k-1 & \text{if } m \in [\![f_{k-1}]\!]_P \wedge m \notin [\![f_k]\!]_P \\ \quad\cdots & \\ k-i & \text{if } m \in [\![f_{k-i}]\!]_P \wedge m \notin \bigcup_{k \geq j > k-i} [\![f_j]\!]_P \\ \quad\cdots & \\ 1 & \text{if } m \in [\![f_1]\!]_P \wedge m \notin \bigcup_{k \geq j > 1} [\![f_j]\!]_P \\ 0 & \text{otherwise} \end{cases}$$

Given P, the parameterized heuristic assigns an abstraction level j to each program component, where the level j is determined according to the model parameter Π. A program component m is assigned the level j if the j-th boolean formula f_j of Π includes the component m, i.e., $m \in [\![f_j]\!]_P$, and m is not implied by any other formulas $f_{j+1}, f_{j+2}, \ldots, f_k$ at higher levels. That is, when m belongs to both f_i and f_j ($i > j$), we favor assigning the greater abstraction level i to m.

4.2 The Optimization Problem

Once we define a model $\mathcal{H}_\Pi$, learning a good abstraction-selection heuristic corresponds to finding a good model parameter Π. Given a codebase $\mathbf{P} = \{P_1, \ldots, P_m\}$ and the model $\mathcal{H}_\Pi$, we define the learning problem as the following optimization problem:

$$\text{Find } \Pi \text{ that minimizes } \sum_{P \in \mathbf{P}} \mathsf{cost}(F_P(\mathcal{H}_\Pi(P))) \text{ while satisfying } \frac{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma. \tag{2}$$

That is, we aim to find a parameter Π that minimizes the cost of the analysis over the codebase while satisfying the precision constraint, $\frac{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$, which asserts that the ratio of the number of assertions proved by the analysis with Π to the number of assertions proved by the most precise analysis must be higher than a predefined threshold $\gamma \in [0, 1]$. For instance, setting γ to 0.9 means that we would like to ensure 90% of the full precision.

Although we assume a single client (e.g. safety of type casts) for presentation brevity, the optimization problem can be defined for multiple clients. Suppose we have n clients, each of which is accompanied with the corresponding projection function $\mathsf{proved}_i (1 \leq i \leq n)$. Then, we can redefine the precision constraint by, for example, $\frac{1}{n} \sum_{j=1}^{n} \frac{\sum_{P \in \mathbf{P}} |\mathsf{proved}_j(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}_j(F_P(\mathbf{k}))|} \geq \gamma$, where we evaluate the overall performance by averaging the results.

4.3 The Learning Algorithm

Note that solving the optimization problem in Equation (2) is extremely challenging. This is mainly because the space of parameters is intractably large. A model parameter Π consists of k boolean formulas. Assuming that $\mathbb{S}$ is the space of possible boolean formulas over which we learn, searching for k formulas simultaneously poses the huge search space of size $|\mathbb{S}|^k$. This space is typically too large to enable effective learning even for small k.

Overall Algorithm. We present a learning algorithm that systematically searches for the best solution. To do so, we first decompose the optimization problem in Equation (2) into k sub-problems: $\Psi_k, \Psi_{k-1}, \ldots, \Psi_1$, which reduces the size of the search space from $|\mathbb{S}|^k$ to $k \cdot |\mathbb{S}|$. Note that the solution of the original problem is a vector of k boolean formulas: $\Pi = \langle f_1, \ldots, f_k \rangle$. In our approach, solving the sub-problem Ψ_i $(1 \leq i \leq k)$ produces the i-th boolean formula f_i of Π. Therefore, we solve the problems Ψ_i $(1 \leq i \leq k)$ separately and combine their solutions f_i $(1 \leq i \leq k)$ to form $\Pi = \langle f_1, \ldots, f_k \rangle$.

The solution f_i for the problem Ψ_i is defined in terms of $f_{i+1}, f_{i+2}, \ldots, f_k$, i.e., the solutions of the problems $\Psi_{i+1}, \Psi_{i+2}, \ldots, \Psi_k$ at higher levels. Suppose we already solved the problems $\Psi_{i+1}, \Psi_{i+2}, \ldots, \Psi_k$ and have their solutions $f_{i+1}, f_{i+2}, \ldots, f_k$. Then, the problem Ψ_i is defined as follows:

$$\Psi_i \equiv \text{ Find } f \text{ that minimizes } \sum_{P \in \mathbf{P}} \text{cost}(F_P(\mathcal{H}_{\Pi_i}(P))) \text{ while satisfying } \frac{\sum\limits_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi_i}(P)))|}{\sum\limits_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma. \tag{3}$$

where $\Pi_i = \langle \mathit{true}, \ldots, \mathit{true}, f, f_{i+1}, f_{i+2}, \ldots, f_k \rangle$. That is, when we solve the problem Ψ_i, we fix the currently available solutions $f_{i+1}, f_{i+2}, \ldots, f_k$ and attempt to find a formula f that achieves the best performance with respect to $f_{i+1}, f_{i+2}, \ldots, f_k$. Note that the first $i-1$ formulas of Π_i are *true*; according to the definition of $\mathcal{H}_\Pi$, this means that we apply the abstraction level $i-1$ to all remaining methods that are not selected by $f, f_{i+1}, f_{i+2}, \ldots, f_k$.

Since solving the problem Ψ_i requires to solve the higher-level problems Ψ_j ($j > i$), we proceed in decreasing order from k to 1: We first solve the problem Ψ_k and use the result when we solve the problem Ψ_{k-1}, and so on. Let f_i be the solution of the problem Ψ_i ($1 \leq i \leq k$). Then, the solution Π of the original problem in Equation (2) is simply obtained by combining the solutions f_i: $\Pi = \langle f_1, f_2, \ldots, f_k \rangle$.

Algorithm 1 presents the learning algorithm. It takes as input static analyzer F, codebase $\mathbf{P}$, the degree of abstraction k, and atomic features $\mathbb{A}$. A vector $\langle f_1, f_2, \ldots, f_k \rangle$ of boolean formulas is returned. At line 2, the formulas are initialized with *true*. At lines 3–5, it iterates the abstraction degrees $k, k-1, \ldots, 1$ in decreasing order and updates the boolean formula f_i of the current depth i. The update is done by invoking the function LearnBooleanFormula, which we describe shortly.

Property of Our Algorithm. Before explaining how LearnBooleanFormula works, we point out that while our learning approach reduces the search space significantly, it does not lose a chance of finding good solutions. Specifically, our algorithm guarantees to preserve a *minimal* solution of the original problem (2). Let us first define the notion of minimal solutions.

Definition 4.1. Let $\mathbf{P}$ be a codebase and $\Pi = \langle f_1, f_2, \ldots, f_k \rangle$ be a parameter. We say Π is a minimal solution of the problem (2) if

(1) Π meets the precision constraint: $\frac{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$, and

(2) there exists no solution smaller than Π: if Π' is a parameter that meets the precision constraint, i.e., $\frac{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'}(P)))|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$, and Π' is smaller than Π, i.e., $\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'}(P) \sqsubseteq \mathcal{H}_\Pi(P)$, then Π' and Π are equivalent:
$$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'}(P) = \mathcal{H}_\Pi(P).$$

In a similar way, we can define the notion of minimal solutions for the sub-problems as follows:

Definition 4.2. Let $\mathbf{P}$ be a codebase and f_i be the solution of the problem Ψ_i. Let Π_i be the vector

$$\langle \mathit{true}, \ldots, \mathit{true}, f_i, f_{i+1}, \ldots, f_k \rangle$$

where $f_{i+1}, \ldots, f_k$ are solutions of problems $\Psi_{i+1}, \ldots, \Psi_k$, respectively. We say f_i is minimal if

(1) Π_i meets the precision constraint: $\frac{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi_i}(P)))|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$, and

(2) Π_i is minimal: if $\Pi'_i = \langle \mathit{true}, \ldots, \mathit{true}, f'_i, f_{i+1}, \ldots, f_k \rangle$ is a parameter that meets the precision constraint, i.e., $\frac{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'_i}(P)))|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$, and Π'_i is smaller than Π_i, i.e., $\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_i}(P) \sqsubseteq \mathcal{H}_{\Pi_i}(P)$, then Π'_i and Π_i are equivalent:
$$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_i}(P) = \mathcal{H}_{\Pi_i}(P).$$

Algorithm 1 Our Learning Algorithm

```
Input: Static analyzer $F$, codebase $\mathbf{P}$, abstraction degree $k$, atomic features $\mathbb{A}$
Output: A vector $\langle f_1, f_2, \ldots, f_k \rangle$ of $k$ boolean formulas
procedure LEARN($F, \mathbf{P}, k$)
    $\langle f_1, f_2, \ldots, f_k \rangle \leftarrow \langle true, true, \ldots, true \rangle$        ▷ initialize $f_1, f_2, \ldots, f_k$ with true
    for $i = k$ to 1 do
        $f_i \leftarrow$ LEARNBOOLEANFORMULA($i, \langle f_1, f_2, \ldots, f_k \rangle, F, \mathbf{P}, \mathbb{A}$)        ▷ update $f_i$
    end for
    return $\langle f_1, f_2, \ldots, f_k \rangle$
end procedure
```

Theorem 4.3 below states that our stepwise learning algorithm is able to produce a minimal solution of the original problem if each formula f_i is a minimal solution of the problem Ψ_i.

Theorem 4.3. *Let $f_1, \ldots, f_k$ be minimal solutions of the problems $\Psi_1, \ldots, \Psi_k$. Then, $\langle f_1, \ldots, f_k \rangle$ is a minimal solution of the original problem (2).*

Proof. See Appendix A.1. □

Learning Boolean Formulas. Now we explain LEARNBOOLEANFORMULA, which is used to solve each sub-problem Ψ_i. Note that the search space of the sub-problem Ψ_i is still huge; there are 2^{2^n} semantically different boolean functions over n boolean variables (i.e., the number of atomic features $\mathbb{A} = \{a_1, \ldots, a_n\}$). Therefore, it is intractable to exhaustively search for a good solution. To address this challenge, we developed a greedy search algorithm that produces good-enough solutions in practice.

Algorithm 2 presents our algorithm for learning a boolean formula f_i for each problem Ψ_i. The algorithm takes as input the current abstraction level i, current formulas $\langle f_1, \ldots, f_k \rangle$, static analyzer F, codebase $\mathbf{P}$, and atomic features $\mathbb{A} = \{a_1, \ldots, a_n\}$. When the algorithm is used for solving the i-th problem (i.e., Ψ_i), we assume that the solutions $f_{i+1}, f_{i+2}, \ldots, f_k$ of the problems $\Psi_{i+1}, \Psi_{i+2}, \ldots, \Psi_k$ are already computed (this is ensured by Algorithm 1).

Given these inputs, the algorithm produces as output a boolean formula f in disjunctive normal form (DNF); f is a disjunction of conjunctions of literals:

$$f = \bigvee_x \bigwedge_y l_{x,y}$$

where a literal $l_{x,y}$ includes boolean constants, atomic features $a_j \in \mathbb{A}$, and their negations $\neg a_j$. In the algorithm, we represent a conjunctive clause (i.e., a conjunction of literals) by a set of literals, and a disjunction by a set of clauses.

At line 3, the algorithm initializes the formula f with a disjunction of all atomic features and their negations:

$$f = a_1 \lor a_2 \lor \cdots \lor a_n \lor \neg a_1 \lor \neg a_2 \lor \cdots \lor \neg a_n$$

Note that this formula denotes the set of all program components in the program, and therefore the initial formula leads to the most precise analysis that assigns the abstraction level i to every component (except for the those already selected by $f_{i+1}, \ldots, f_k$). Beginning with this formula f, the goal of our algorithm is to refine each clause of f and obtain a boolean formula that minimizes the analysis cost while preserving the precision constraint (e.g., achieving 90% of the full precision).

To do so, the algorithm maintains a workset W which is a set of clauses to refine further. The workset initially contains all atomic clauses (line 4). The algorithm iterates while the workset is

Algorithm 2 Algorithm for Learning a Boolean Formula

```
Input: Abstraction level $i$, current formulas $\langle f_1, f_2, \ldots, f_k \rangle$, static analyzer $F$, codebase $\mathbf{P}$, atomic
    features $\mathbb{A}$
Output: Boolean formula $f_i$ in disjunctive normal form
 1: procedure LearnBooleanFormula($i, \langle f_1, f_2, \ldots, f_k \rangle, F, \mathbf{P}, \mathbb{A}$)
 2:     $\mathbb{A}' \leftarrow \mathbb{A} \cup \{\neg a_j \mid a_j \in \mathbb{A}\}$        ▷ atomic features and their negations
 3:     $f \leftarrow \{\{a_j\} \mid a_j \in \mathbb{A}'\}$        ▷ initial formula
 4:     $W \leftarrow f$        ▷ initial workset (the set of all clauses in $f$)
 5:     $bestCost \leftarrow \infty$        ▷ initial best cost
 6:     while $W \neq \emptyset$ do
 7:         $c \leftarrow \text{ChooseClause}(W, F, \mathbf{P})$        ▷ choose the most expensive clause from $W$
 8:         $W \leftarrow W \setminus \{c\}$
 9:         $a \leftarrow \text{ChooseAtom}(\mathbb{A}', c, F, \mathbf{P})$        ▷ choose an atom from $\mathbb{A}'$
10:         $c' \leftarrow c \cup \{a\}$        ▷ refined clause
11:         $f' \leftarrow (f \setminus \{c\}) \cup \{c'\}$        ▷ refined formula
12:         $\Pi \leftarrow \langle f_1, \ldots, f_{i-1}, f', f_{i+1}, f_{i+2}, \ldots, f_k \rangle$        ▷ current parameter setting
13:         $(proved, cost) \leftarrow \text{Analyze}(\Pi, F, \mathbf{P})$
14:         if $cost \leq bestCost \land \frac{|proved|}{\sum_{P \in \mathbf{P}} |\text{proved}(F_P(\mathbf{k}))|} \geq \gamma$ then        ▷ cheaper parameter found
15:             $bestCost \leftarrow cost$        ▷ update the best cost
16:             if $(f' \iff f \setminus \{c\})$ then        ▷ check if $f'$ is semantically refined
17:                 $f \leftarrow f' \setminus \{c'\}$        ▷ remove chosen clause from $f$
18:                 continue
19:             else
20:                 $W \leftarrow W \cup \{c'\}$        ▷ $c'$ can be refined further
21:                 $f \leftarrow f'$        ▷ update the formula
22:             end if
23:         end if
24:     end while
25:     return $f$
26: end procedure
```

non-empty. At lines 7 and 8, a clause is selected and removed from the workset. Our algorithm is greedy in a sense that the ChooseClause function chooses the most expensive clause c from W:

$$\text{ChooseClause}(W, F, \mathbf{P}) = \underset{c \in W}{\text{argmax}} \sum_{P \in \mathbf{P}} \text{cost}(F_P(\mathcal{H}_{\Pi_c}(P)))$$

where $\Pi_c = \langle f_1, \ldots, f_{i-1}, c, f_{i+1}, f_{i+2}, \ldots, f_k \rangle$. The heuristic, $\mathcal{H}_{\Pi_c}$, with Π_c assigns the abstraction level i to the components for which c is true (except for those for which some of $f_{i+1}, \ldots, f_k$ are true). All the other components are assigned the depth $i - 1$, because LearnBooleanFormula is invoked with $f_1, \ldots, f_{i-1}$ being *true*.

The next step is to refine the clause c by conjoining an atom $a \in \mathbb{A}'$ to c (lines 9 and 10): i.e., $c' = c \land a$. The refined clause c' represents a smaller set of components than c, which decreases the precision of the analysis. When refining the clause, our algorithm is conservative and chooses the atom $a \in \mathbb{A}'$ with which refining c decreases the analysis precision as little as possible. More precisely, the ChooseAtom function is defined as follows:

$$\mathsf{ChooseAtom}(\mathbb{A}', c, F, \mathbf{P}) = \begin{cases} \mathrm{argmax}_{a \in \mathbb{A}' \setminus c} \sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi_{a \wedge c}}(P)))| & \text{if } \mathbb{A}' \setminus c \neq \emptyset \\ \mathit{false} & \text{otherwise} \end{cases}$$

where $\Pi_{a \wedge c} = \langle f_1, \ldots, f_{i-1}, a \wedge c, f_{i+1}, f_{i+2}, \ldots, f_k \rangle$. When there exists an atom to choose (i.e., $\mathbb{A}' \setminus c \neq \emptyset$), we conservatively choose the atom a with the greatest precision. Otherwise, there is no atom to refine with and *false* is returned so that the clause c does not get refined further. In the latter case, the algorithm eventually goes to line 18 (because $f' \iff f \setminus \{c\}$ is valid) and attempts to choose another clause to refine. When an atom a is successfully chosen, we refine the clause (line 10) and the formula (line 11).

At lines 12–13, the refined formula is evaluated. We first construct the parameter setting Π with the current formula f' (line 12):

$$\Pi = \langle f_1, \ldots, f_{i-1}, f', f_{i+1}, f_{i+2}, \ldots, f_k \rangle.$$

Next, we analyze the programs in the codebase with Π. The Analyze function returns the set of queries proved and the cost spent with the parameter Π:

$$\mathsf{Analyze}(\Pi, F, \mathbf{P}) = (\sum_{P \in \mathbf{P}} \mathsf{proved}(F_P(\mathcal{H}_\Pi(P))), \sum_{P \in \mathbf{P}} \mathsf{cost}(F_P(\mathcal{H}_\Pi(P)))).$$

At line 14, we check whether the cost is actually reduced while ensuring the precision constraint. If so, *bestCost* is updated with the current cost. At line 16, we check if the rest clauses of the old formula ($f \setminus \{c\}$) cover the refined clause c'. If so, we remove the clause c' from the formula (line 17) and try to refine another clause. For instance, suppose f is $a_1 \vee a_2 \vee a_3$ and it is refined to $f' = a_1 \vee (a_1 \wedge a_2) \vee a_3$. We remove the refined clause $a_1 \wedge a_2$ because $a_1 \wedge a_2 \implies a_1$. If the condition at line 16 is false, we update the workset with the refined clause c' (i.e., c' can be refined further) and f gets replaced by f'. If the performance is not improved or the precision constraint is violated, we do not add the refined clause c' to the workset and f does not get updated.

Termination and Complexity. Note that the algorithm is guaranteed to terminate. First of all, the workset W never grows in each iteration of the loop. After a clause is removed from the workset at line 8, the algorithm either goes into the next iteration (line 18) or refines the clause and pushes it back to the workset (line 20). Furthermore, a clause never gets endlessly refined during the algorithm. Once a clause becomes a conjunction of all atoms, the ChooseAtom function returns *false* which makes the condition at line 16 true and that clause is permanently removed from the workset. Therefore, the workset eventually becomes empty in finite steps. The algorithm has the asymptotic upper bound of $O(k \times |A|^2)$, where k is the maximally allowed context depth and A is the set of atomic features.

5 APPLICATION TO CONTEXT-SENSITIVE POINTS-TO ANALYSIS FOR JAVA

In this article, we apply our data-driven approach to two program analyses: context-sensitive points-to analysis for Java and flow-sensitive interval analysis for C. In this section, we describe the first instance analysis.

To use our approach, we need to define a parametric static analysis and atomic features. Section 5.2 defines our parametric context-sensitive analysis and Section 5.3 presents the atomic features we used. We build on the previous work [Kastrinis and Smaragdakis 2013c] that defines a generic context-sensitive points-to analysis in Datalog (Section 5.1). We incrementally extend the analysis to allow different context depths for each method. This section will use the same notations introduced by Kastrinis and Smaragdakis [2013c].

Input Relations

Alloc (*var*: V, *heap*: H, *inMeth*: M)
Move($to : V, from : V$)
Load($to : V, base : V, fld : F$)
Store($base : V, fld : F, from : V$)
VCall($base : V, sig : S, invo : I, inMeth : M$)
SCall($meth : M, invo : I, inMeth : M$)

FormalArg($meth : M, i : \mathbb{N}, arg : V$)
ActualArg($invo : I, i : \mathbb{N}, arg : V$)
FormalReturn($meth : M, ret : V$)
ActualReturn($invo : I, var : V$)
ThisVar($meth : M, this : V$)
HeapType($heap : H, type : T$)
LookUp($type : T, sig : S, meth : M$)

Output Relations

VarPointsTo($var : V, ctx : C, heap : H, hctx : HC$)
CallGraph($invo : I, callerCtx : C, meth : M, calleeCtx : C$)
FldPointsTo($baseH : H, baseHCtx : HC, fld : F, heap : H, hctx : HC$)
InterProcAssign($to : V, toCtx : C, from : V, fromCtx : C$)
Reachable($meth : M, ctx : C$)

Fig. 3. Input and output relations of points-to analysis from [Kastrinis and Smaragdakis 2013c]

5.1 Points-to Analysis by Kastrinis and Smaragdakis [2013c]

We summarize the parametric points-to analysis designed by Kastrinis and Smaragdakis [2013c]. For more details, we refer the readers to prior work [Kastrinis and Smaragdakis 2013c; Smaragdakis and Balatsouras 2015].

In [Kastrinis and Smaragdakis 2013c], a Java program is represented as Datalog relations shown in Fig. 3. Input relations are grouped into instructions and auxiliary information. The meaning of the instructions is straightforward. For instance, Alloc relation models a heap allocation, where V, H, and M denote the sets of program variables, heap abstractions (i.e., allocation-sites), and method identifiers, respectively. F, S, and I denote the sets of fields, method signatures, and instructions, respectively. The auxiliary relations encode the name and type information. For instance, FormalArg encodes that *arg* is the i-th formal argument of *meth* (resp., the method at *invo*).

Given the input relations, the analysis derives the output relations listed in the bottom of Fig. 3. The VarPointsTo and CallGraph relations represent results of the context-sensitive points-to analysis. The former describes that the variable *var* in the call context *ctx* may points to the heap location *heap* whose heap context is *hctx*. Likewise, CallGraph(*invo*, *callerCtx*, *meth*, *calleeCtx*) encodes the context-sensitive call graph: the method *meth* can be invoked at the instruction *invo* with respect to the caller and callee contexts: *callerCtx* and *calleeCtx*.

Fig. 4(a) shows the points-to analysis rules used by Kastrinis and Smaragdakis [2013c], which performs a flow-insensitive and context-sensitive points-to analysis with on-the-fly call-graph construction. The rules specify, for each instruction type, how to derive the output relations from the input relations. For instance, the fourth rule corresponds to the copy instruction.

The most important feature of the analysis is that context-sensitivity is encapsulated by the following three constructor functions:

- **Record**($heap : H, ctx : C$) produces new heap contexts. It is used when allocating heap objects (i.e., Alloc) and creates new heap contexts for them. Given an allocation-site and a calling-context, **Record** returns a new heap context for the heap object.

```
InterProcAssign(to, calleeCtx, from, callerCtx) ←
  CallGraph(invo, callerCtx, meth, calleeCtx), FormalArg(meth, i, to),
  ActualArg(invo, i, from).

InterProcAssign(to, callerCtx, from, calleeCtx) ←
  CallGraph(invo, callerCtx, meth, calleeCtx), FormalReturn(meth, from),
  ActualReturn(invo, to).

Record(heap, ctx)=hctx, VarPointsTo(var, ctx, heap, hctx) ←
  Reachable(meth, ctx), Alloc(var, heap, meth).

VarPointsTo(to, ctx, heap, hctx) ← Move(to, from), VarPointsTo(from, ctx, heap, hctx).

VarPointsTo(to, toCtx, heap, hctx) ←
  InterProcAssign(to, toCtx, from, fromCtx), VarPointsTo(from, fromCtx, heap, hctx).

VarPointsTo(to, ctx, heap, hctx) ←
  Load(to, base, fld), VarPointsTo(base, ctx, baseH, baseHCtx),
  FldPointsTo(baseH, baseHCtx, fld, heap, hctx).

FldPointsTo (baseH, baseHCtx, fld, heap, hctx) ←
  Store(base, fld, from), VarPointsTo(from, ctx, heap, hctx),
  VarPointsTo(base, ctx, baseH, baseHCtx).

Merge (heap, hctx, invo, callerCtx)=calleeCtx,
Reachable (toMeth, calleeCtx),
VarPointsTo (this, calleeCtx, heap, hctx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ←
  VCall (base, sig, invo, inMeth),
  Reachable (inMeth, callerCtx), VarPointsTo (base, callerCtx, heap, hctx),
  HeapType (heap, heapT), LookUp (heapT, sig, toMeth), ThisVar (toMeth, this).

MergeStatic (invo, callerCtx) = calleeCtx,
Reachable (toMeth, calleeCtx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ←
  SCall (toMeth, invo, inMeth), Reachable (inMeth, callerCtx).
```

(a) Points-to analysis rules taken from [Kastrinis and Smaragdakis 2013c]

```
Merge (depth, heap, hctx, invo, callerCtx)=calleeCtx,
Reachable (toMeth, calleeCtx),
VarPointsTo (this, calleeCtx, heap, hctx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ←
  VCall (base, sig, invo, inMeth), Reachable (inMeth, callerCtx),
  VarPointsTo (base, callerCtx, heap, hctx), HeapType (heap, heapT),
  LookUp (heapT, sig, toMeth), ThisVar (toMeth, this), ApplyDepth (toMeth, depth).

MergeStatic (depth, invo, callerCtx) = calleeCtx,
Reachable (toMeth, calleeCtx),
CallGraph (invo, callerCtx, toMeth, calleeCtx) ←
  SCall (toMeth, invo, inMeth), Reachable (inMeth, callerCtx),
  ApplyDepth (toMeth, depth).
```

(b) Modified rules for our parametric points-to analysis

Fig. 4. Datalog rules for context-sensitive points-to analysis

- $\textsc{Merge}(heap : H, hctx : HC, invo : I, ctx : C)$ creates calling contexts for virtual calls. Given heap object, heap context, call-site, and calling context, it creates a new context for called functions.
- $\textsc{MergeStatic}(invo : I, ctx : C)$ is similar to **Merge** but it is used for static method calls. Given a method call with a calling context, it creates a new calling context.

Kastrinis and Smaragdakis [2013c] showed that a large class of context-sensitive analyses (including k-call-site sensitivity, k-object-sensitivity, k-type-sensitivity, and their variants) can be obtained by appropriately defining the constructor functions and the domains (HC and C). Below, we define three types of points-to analyses that we consider in this paper.

- 2-object-sensitive with a 1-context-sensitive heap (*2objH*). We get this standard object-sensitivity by using allocation-sites as heap contexts (i.e., $HC = H$) and two allocation-sites as calling contexts (i.e., $C = H \times H$). The definitions of the constructor functions are as follows:

$$\begin{array}{rcl} \textsc{Record}(heap, ctx) & = & first(ctx) \\ \textsc{Merge}(heap, hctx, invo, ctx) & = & pair(heap, hctx) \\ \textsc{MergeStatic}(invo, ctx) & = & ctx \end{array}$$

  At virtual method calls (**Merge**), the context is created by appending the receiver object (*heap*) and its heap context (*hctx*). Note that **Record** uses the first element of *ctx*; the new heap context of an object is the receiver object of the allocating method. At static calls (**MergeStatic**), the calling context of the caller method is used without changes.
- Selective 2-object-sensitive with 1-object-sensitive heap hybrid (*S2objH*) [Kastrinis and Smaragdakis 2013c]. Kastrinis and Smaragdakis [2013c] proposed a variant of object-sensitivity that selectively uses object-sensitivity for virtual calls and call-site-sensitivity for static calls:

$$\begin{array}{rcl} \textsc{Record}(heap, ctx) & = & first(ctx) \\ \textsc{Merge}(heap, hctx, invo, ctx) & = & triple(heap, hctx, \star) \\ \textsc{MergeStatic}(invo, ctx) & = & triple(first(ctx), invo, second(ctx)) \end{array}$$

  where $HC = C$ and $C = H \times (H \cup I) \times (H \cup I \cup \{\star\})$. Note that **Record** and **Merge** are essentially the same as those for the standard object-sensitivity. For static calls, the analysis keeps call-sites (*invo*) and the partial context information (*first*(*ctx*)) of the caller, which makes the precision incomparable to that of *2objH*.
- 2-type-sensitive with a 1-context-sensitive heap (*2typeH*) [Smaragdakis et al. 2011]. Type-sensitivity is an abstraction of object-sensitivity. While object-sensitivity uses an allocation-site as contexts, type-sensitivity uses the name of the class enclosing the allocation-site. Therefore, the definition of type-sensitivity can be obtained by slightly changing the definition of object-sensitivity as follows:

$$\begin{array}{rcl} \textsc{Record}(heap, ctx) & = & first(ctx) \\ \textsc{Merge}(heap, hctx, invo, ctx) & = & pair(ClassOf(heap), hctx) \\ \textsc{MergeStatic}(invo, ctx) & = & ctx \end{array}$$

  where *ClassOf*(*heap*) gets the name of the class that contains the allocation-site *heap*.

5.2 Extension to Our Parametric Analysis

We extend the analysis rules in Fig. 4(a) to assign different context depths to different methods (in a similar way to the parametric framework by Milanova et al. [2005]). For this purpose, we extend the prior analysis in two ways. First, our analysis requires the extra input relation:

$$\textsc{ApplyDepth}(meth : M, depth : \mathbb{N}).$$

The **ApplyDepth** relation maps methods to their context depths; the method (*meth*) is analyzed with the given context-sensitivity depth (*depth*). In this section, we assume that the mapping (i.e., a set of **ApplyDepth** relations) is given for the target program. In our approach, the heuristic that we learn from codebases generates the relations.

Second, we need to modify the context constructors **Merge** and **MergeStatic** so that they produce new contexts by considering the given context depths as well:

$$\textbf{Merge}(\mathit{depth} : \mathbb{N}, \mathit{heap} : H, \mathit{hctx} : HC, \mathit{invo} : I, \mathit{ctx} : C) = \mathit{newCtx} : C$$
$$\textbf{MergeStatic}(\mathit{depth} : \mathbb{N}, \mathit{invo} : I, \mathit{ctx} : C) = \mathit{newCtx} : C$$

With these new constructors, we replace the last two rules in Fig. 4(a) by the rules in Fig. 4(b). For instance, a virtual method call VCall(*base*, *sig*, *invo*, *inMeth*) is handled as follows:

(1) VarPointsTo figures out a set of *heaps* that the *base* can point to.
(2) From each *heap*, a type identifier *heapT* is revealed.
(3) Using the identifier and the invocation's signature, the target method *toMeth* is found.
(4) ApplyDepth returns *depth* according to the *toMeth*, and it is provided to the Merge constructor.

MergeStatic is defined in a similar way but, in this case, SCall itself has the *toMeth* information. The other seven rules in Fig. 4(a) are used without changes.

All existing context-sensitive analyses expressible by the previous framework [Kastrinis and Smaragdakis 2013c] can be easily extended to our framework. For instance, consider the *2objH* analysis. We use the same definition for *HC* while *C* is modified to allow shallower depths; $C = (H \cup \{\star\}) \times (H \cup \{\star\})$ is the new context type. With these domains, the constructor functions are defined as follows:

$$\begin{aligned}
\textbf{Record}(\mathit{heap}, \mathit{ctx}) &= \mathit{first}(\mathit{ctx}),\\
\textbf{Merge}(\mathit{depth}, \mathit{heap}, \mathit{hctx}, \mathit{invo}, \mathit{ctx}) &= \begin{cases} \mathit{pair}(\mathit{heap}, \mathit{hctx}) & \text{if } \mathit{depth} = 2\\ \mathit{pair}(\mathit{heap}, \star) & \text{if } \mathit{depth} = 1\\ \mathit{pair}(\star, \star) & \text{if } \mathit{depth} = 0 \end{cases}\\
\textbf{MergeStatic}(\mathit{depth}, \mathit{invo}, \mathit{ctx}) &= \begin{cases} \mathit{pair}(\mathit{first}(\mathit{ctx}), \mathit{second}(\mathit{ctx})) & \text{if } \mathit{depth} = 2\\ \mathit{pair}(\mathit{first}(\mathit{ctx}), \star) & \text{if } \mathit{depth} = 1\\ \mathit{pair}(\star, \star) & \text{if } \mathit{depth} = 0 \end{cases}
\end{aligned}$$

When $\mathit{depth} = 2$, note that the analysis is identical to *2objH*. When $\mathit{depth} = 1$, the **Merge** and **MergeStatic** truncate the contexts and maintain only the last context element (i.e., *1objH*). When *depth* is 0, the method is analyzed with context-insensitivity. We can use the same principle to transform other analyses such as *S2objH* and *2typeH* to our parametric setting.

5.3 Atomic Features

Table 1 shows the atomic features for the context-sensitive points-to analysis for Java. In any application of machine learning, the success depends heavily on the quality of the features. For instance, in the existing approach by Oh et al. [2015], authors manually crafted 45 high-level features for program variables, which are then used for learning to apply flow-sensitivity in interval analysis. However, coming up with such high-quality features manually is a nontrivial task requiring a large amount of engineering effort and domain expertise.

In this work, to reduce the feature-engineering burden, we focus on generating only low-level and easy-to-obtain atomic features and utilize a learning algorithm to synthesize high-level features. We used features found in signature and body of a method which are readily obtainable from any Java frontend such as Soot [Vallée-Rai et al. 1999].

Table 1. Atomic features

Signature features									
#1	"java"	#3	"sun"	#5	"void"	#7	"int"	#9	"String"
#2	"lang"	#4	"()"	#6	"security"	#8	"util"	#10	"init"

Statement features					
#11	AssignStmt	#16	BreakpointStmt	#21	LookupStmt
#12	IdentityStmt	#17	EnterMonitorStmt	#22	NopStmt
#13	InvokeStmt	#18	ExitMonitorStmt	#23	RetStmt
#14	ReturnStmt	#19	GotoStmt	#24	ReturnVoidStmt
#15	ThrowStmt	#20	IfStmt	#25	TableSwitchStmt

Using Soot, we generated two types of atomic features: features for method signatures and features for statements. A signature feature describes whether the method's signature contains a particular string. For instance, the first feature in Table 1 indicates whether the method contains string "java" in its signature. From a training program, we generated all words contained in method signatures and collected the top-10 words that most frequently appear. Features #1–10 show the signature features generated this way. A statement feature indicates whether the method has a particular type of statements. We used 15 statement types available in Soot (#11–25 in Table 1). For instance, the feature #11 indicates whether the method has at least one assignment statement in its body. Combining the types of features, we generated 25 atomic features.

Regarding signature features, we chose top-10 features because they provide enough frequency spectrum, both general and specific. For instance, the feature #1 ("java") appeared 142,097 times over the training programs, whereas the feature #10 ("init") appeared 31,984 times. First five features are general method properties, and the others are specific ones. Both of general and specific features are needed to generate accurate yet generalizable context-selection heuristics. For example, application of features #1–#5, without specific features #6–#10, makes our algorithm to fail to find a cost-effective heuristic. Inclusion of specific features allow our analysis to become more efficient without significant trade-off on precision on analysis result. Without features #6–#10, timeout would occur on large programs. In Section 7.1, we provide more detailed discussion with experimental results.

Our learning approach works well without high-level features, mainly because the learning model (i.e., parameterized heuristic) is powerful and able to automatically generate those features by combining the atomic features via boolean formulas. On the other hand, the learning model used by Oh et al. [2015] has limited expressiveness; the model combines the features by simple linear combination, which cannot express, for instance, disjunctions of atomic features.

6 APPLICATION TO FLOW-SENSITIVE INTERVAL ANALYSIS FOR C

The second application of our approach is an interval analysis for C programs. We build on the previous work [Oh et al. 2015] that defines a selective flow-sensitive interval analysis based on sparse analysis [Oh et al. 2012]. This section will use the same notations introduced by Oh et al. [2015]. Section 6.2 presents the features we used for this analysis instance.

6.1 Selective Flow-Sensitive Analysis

Given a program P, let $(\mathbb{C}, \hookrightarrow)$ be its control flow graph, where $\mathbb{C}$ is the set of nodes (program points) and $(\hookrightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ denotes the control flow relation of the program.

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

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

where an abstract state $s \in \mathbb{S}$ is a mapping from abstract locations (e.g. program variables, structure fields, and allocation sites) to values:

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

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

As in [Oh et al. 2015], we consider an extension of the sparse-analysis framework [Oh et al. 2012]. Let $\mathsf{D}(c) \subseteq \mathbb{L}$ and $\mathsf{U}(c) \subseteq \mathbb{L}$ be the definition and use sets at program point $c \in \mathbb{C}$. Using these sets, we define a data-dependency relation $(\rightsquigarrow) \subseteq \mathbb{C} \times \mathbb{L} \times \mathbb{C}$ as follows:

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

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

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

We say this analysis is fully flow-sensitive because it constructs data dependencies for all abstract locations and tracks all of these dependencies accurately.

We extend this sparse-analysis into an analysis that is allowed to track data dependencies only for a subset of abstract locations in some set $L \subseteq \mathbb{L}$, and to be flow-sensitive only for these locations. The remaining locations (i.e., $\mathbb{L} \setminus L$) are analyzed flow-insensitively; we use results from a quick flow-insensitive pre-analysis [Oh et al. 2012], which we assume given. The results of this pre-analysis form a state $s_I \in \mathbb{S}$, and are stable (i.e., pre-fixpoint) at all program points:

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

Next, we define the partial data-dependency with respect to L:

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

In this definition, we require that in order for $c_0 \overset{l}{\rightsquigarrow}_L c_n$ to hold, the location l should be included in the set L. With this notion of partial data-dependency, we define the abstract transfer function as follows:

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

This definition says that when we collect an abstract state right before c, we use the flow-insensitive result $s_I(l)$ for a location not in L, and follow the original treatment for those in L. An analysis in our extension computes $\mathsf{lfp}_{X_0} F_L$, where the initial $X_0 \in \mathbb{D}$ is built by associating the results of the flow-insensitive analysis (i.e., values of s_I) with all locations not selected by L (i.e., $\mathbb{L} \setminus L$):

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

In this analysis, the location set L determines the degree of flow-sensitivity. For instance, when $L = \mathbb{L}$, the analysis denotes an ordinary flow-sensitive analysis. On the other hand, when $L = \emptyset$, the analysis degenerates to a flow-insensitive analysis.

6.2 Atomic Features

We used the same set of 45 features designed for flow-sensitivity by Oh et al. [2015]. Please refer to Table 14 in Appendix A.3 for more details of them. These features describe syntactic or semantic properties of abstract locations, namely, program variables, structure fields and allocation sites, which describe how program locations are used in typical C programs. More details on those features are available from the prior work [Oh et al. 2015].

7 EXPERIMENTS

In this section, we experimentally evaluate our data-driven approach with application to two instance analyses. The main application is context-sensitive points-to analysis for Java, where the objective of the evaluation is to answer the following research questions:

- **Effectiveness and Generalization**: How well does our data-driven approach perform compared to the existing approaches? Does our learning approach generalize well on unseen data? (Section 7.1)
- **Adequacy of Our Learning Algorithm**: Is the disjunctive model essential for learning cost-effective heuristics? How much is it better than the simpler non-disjunctive model? (Section 7.2)
- **Learned Features**: What are the interesting findings on learned boolean formulas? (Section 7.3)

In addition, we show the generality of our learning algorithm by applying it to a flow-sensitive interval analysis for C programs (Section 7.4).

For points-to analysis for Java, we implemented our approach on top of a specific version of Doop, which was used in the PLDI 2014 paper by Smaragdakis et al. [2014]. This version of Doop uses an academic version of the LogicBlox engine, which is a single-thread program. For interval analysis for C, we used Sparrow [Lee et al. 2017; Oh et al. 2014a, 2015], an open-source framework for static analysis of C programs[1]. We used the DaCapo benchmark suite [Blackburn et al. 2006] and open-source C programs to evaluate our approach. All experiments were done on a machine with Intel i5 CPU and 16 GB RAM running on Ubuntu 14.04 64bit operating system and JDK 1.6.0_24.

7.1 Effectiveness and Generalization

Setting. We applied our data-driven approach to three existing context-sensitive points-to analyses: selective 2-object-sensitive (*S2objH*), 2-object-sensitive (*2objH*), and 2-type-sensitive (*2typeH*) analyses, all with 1-context-sensitive heap. All of these analyses are readily available in Doop. *S2objH* and *2objH* are known to be the state-of-the-art points-to analyses for Java with good precision/cost trade-offs [Kastrinis and Smaragdakis 2013b; Milanova et al. 2005]. *2typeH* is another good alternative for precise yet scalable points-to analysis [Smaragdakis et al. 2011]. With our approach, we made data-driven versions of these analyses: *S2objH+Data*, *2objH+Data*, and *2typeH+Data*. In addition, we also made the introspective versions [Smaragdakis et al. 2014] of the three analyses: *S2objH+IntroA*, *S2objH+IntroB*, *2objH+IntroA*, *2objH+IntroB*, *2typeH+IntroA*, and *2typeH+IntroB*. The introspective versions are available in Doop, except for *S2objH+IntroA* and *S2objH+IntroB*. We implemented these two analyses by reusing the code of introspective analysis in Doop.

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

Table 2. Comprehensive performance numbers for all analyses against testing and validation (denoted by *) benchmarks: context-insensitivity (Insens), selective object sensitivity, object-sensitivity, and type-sensitivity. For each analysis, except for Insens, we made four variants: the most precise analyses (*S2objH*, *2objH*, *2typeH*), introspective analyses (IntroA and IntroB), and our data-driven analysis (Ours). In all metrics, lower is better. Entries with dash (-) means the analysis did not finish within time constraint (5400 sec.). For precision metrics, we have the number of virtual calls that points-to analysis cannot uniquely resolve the target, the number of reachable methods, and the number of may-fail casts. For cost metrics, we have the number of call-graph edges and analysis time. The numbers to the right of 'main-fail casts' indicate the total number of assertions, which are reachable during a context-insensitive analysis).

		Insens	Selective object sensitivity				Object-sensitivity				Type-sensitivity			
			S2objH	IntroA	IntroB	Ours	*2objH*	IntroA	IntroB	Ours	*2typeH*	IntroA	IntroB	Ours
eclipse	poly v-calls	1,334	979	1,118	1,045	1,066	980	1,118	1,046	1,060	1,031	1,161	1,100	1,119
	reachable mthds	8,465	7,910	8,216	8,000	7,971	7,911	8,319	8,001	7,959	7,933	8,336	8,026	7,995
	may-fail casts(1,635)	1,139	456	892	676	596	546	977	764	661	665	1,004	850	807
	call-graph-edges	45,474	2.9M	0.8M	1.2M	0.1M	3.4M	0.9M	1.2M	0.1M	0.6M	0.2M	0.4M	59,389
	analysis time(s)	18	79	41	53	23	91	42	52	23	42	39	44	33
chart	poly v-calls	1,852	1,378	1,612	1,512	1,441	1,378	1,613	1,497	1,435	1,446	1,658	1,541	1,516
	reachable mthds	12,064	11,328	11,791	11,589	11,400	11,330	11,952	11,518	11,362	11,439	11,976	11,579	11,474
	may-fail casts(2,586)	1,810	757	1,458	1,191	922	883	1,580	1,236	974	1,147	1,656	1,376	1,245
	call-graph-edges	63,453	8M	2.4M	3.9M	0.1M	9.3M	1.6M	2.9M	0.1M	0.6M	0.3M	0.5M	86,383
	analysis time(s)	34	196	188	213	34	178	91	133	34	60	76	85	62
bloat	poly v-calls	2,014	1,426	1,684	1,521	1,504	1,427	1,690	1,522	1,496	1,626	1,812	1,684	1,680
	reachable mthds	8,939	8,469	8,728	8,625	8,526	8,470	8,869	8,626	8,513	8,523	8,885	8,647	8,564
	may-fail casts(2,436)	1,924	1,125	1,747	1,555	1,232	1,193	1,809	1,621	1,288	1,485	1,832	1,713	1,564
	call-graph-edges	61,150	35M	0.5M	2M	0.2M	35.1M	0.5M	2.0M	0.3M	0.7M	0.1M	0.3M	86,291
	analysis time(s)	22	2,184	39	96	30	2,187	43	96	43	53	44	51	42
xalan	poly v-calls	1,898	1,518	1,743	1,575	1,581	1,522	1,765	1,579	1,583	1,565	1,793	1,640	1,658
	reachable mthds	9,705	9,043	9,365	9,115	9,155	9,047	9,637	9,119	9,142	9,151	9,655	9,232	9,193
	may-fail casts(1,698)	1,182	447	1,055	638	538	533	1,129	723	604	728	1,136	888	812
	call-graph-edges	51,302	9M	1.4M	5.6M	0.1M	11.6M	1.6M	6.8M	0.1M	0.9M	0.3M	0.7M	66,206
	analysis time(s)	29	414	75	329	35	672	78	484	38	71	75	92	63
jython	poly v-calls	2,778	-	2,616	-	2,500	-	2,632	-	2,481	-	2,665	2,479	2,556
	reachable mthds	12,718	-	12,596	-	12,024	-	12,663	-	12,008	-	12,679	12,143	12,048
	may-fail casts(2,790)	2,234	-	2,109	-	1,722	-	2,202	-	1,773	-	2,209	1,984	1,913
	call-graph-edges	0.1M	-	6.6M	-	0.3M	-	6.2M	-	1.1M	-	0.5M	9M	0.1M
	analysis time(s)	73	-	348	-	105	-	353	-	438	-	171	1,443	132
hsqldb*	poly v-calls	1,592	-	1,390	1,257	1,204	-	1,482	1,260	1,195	1,187	1,495	1,288	1,247
	reachable mthds	11,486	-	10,852	10,371	10,395	-	11,367	10,378	10,387	10,333	11,373	10,397	10,438
	may-fail casts(2,229)	1,662	-	1,385	953	1,064	-	1,558	1,034	1,053	1,028	1,578	1,180	1,257
	call-graph-edges	63,790	-	1.4M	10.7M	0.3M	-	1.0M	7.6M	0.7M	1.7M	0.2M	0.5M	0.1M
	analysis time(s)	42	-	77	247	43	-	74	203	261	127	79	78	68

In summary, we compared the performance of the following context-sensitive analyses:

- Selective object-sensitivity:
  - *S2objH*: selective 2-object-sensitivity with 1 context-sensitive heap hybrid [Kastrinis and Smaragdakis 2013b]
  - *S2objH+Data*: our data-driven version of *S2objH*.
  - *S2objH+IntroA*: introspective version of *S2objH* with the Heuristic A [Smaragdakis et al. 2014]
  - *S2objH+IntroB*: introspective version of *S2objH* with the Heuristic B [Smaragdakis et al. 2014]
- Object-sensitivity:
  - *2objH*: 2-object-sensitivity with 1 context-sensitive heap [Kastrinis and Smaragdakis 2013b; Milanova et al. 2005]
  - *2objH+Data*: our data-driven version of *2objH*.
  - *2objH+IntroA*: introspective version of *2objH* with the Heuristic A [Smaragdakis et al. 2014]
  - *2objH+IntroB*: introspective version of *2objH* with the Heuristic B [Smaragdakis et al. 2014]
- Type-sensitivity:
  - *2typeH*: 2-type-sensitivity with 1 context-sensitive heap [Smaragdakis et al. 2011]
  - *2typeH+Data*: our data-driven version of *2typeH*.
  - *2typeH+IntroA*: introspective version of *2typeH* with the Heuristic A [Smaragdakis et al. 2014]
  - *2typeH+IntroB*: introspective version of *2typeH* with the Heuristic B [Smaragdakis et al. 2014]

As it is done by Smaragdakis et al. [2014], we partitioned the ten programs from the DaCapo suite into four small (`antlr`, `lusearch`, `luindex`, and `pmd`) and six large (`eclipse`, `xalan`, `chart`, `bloat`, `hsqldb`, and `jython`) programs. We used the four small programs as a training set where we learned context-selection heuristics. We used `hsqldb` for choosing the value of γ, i.e., the precision threshold of the optimization problem in (2). To choose γ, for each of γ between 0.85 and 0.95 with interval 0.01, we learned from the training set a context-selection heuristic, evaluated its performance on `hsqldb`, and chose γ that shows best performance according to $\frac{\text{\#proved assertions}}{\text{analysis time(s)}}$. The final heuristic with the chosen γ was evaluated on the remaining five test programs (`eclipse`, `xalan`, `chart`, `bloat`, and `jython`). The best γ were 0.93, 0.92, and 0.88 for selective object-sensitivity, object-sensitivity, and type-sensitivity, respectively. We used `hsqldb` for choosing γ since it is one of the two most challenging programs (`jython` and `hsqldb`) in the DaCapo benchmark suite and is suitable for a representative benchmark. Our learning algorithm took 30 hours for learning the depth-2 formula (f_2), and 24 hours for the depth-1 formula (f_1) on the four training programs. In all experiments, we measured the precision of the heuristics using the may-fail casts client, where the assertions can be automatically generated for each casting expression of the program. Lastly, while introspective analysis selects heap allocations as well, we analyzed all heap allocations context-sensitively.

Effectiveness. Fig. 5 compares the performance of our approach for selective object-sensitivity. We discuss the case of selective object-sensitivity in detail, as it is arguably the best context abstraction available to Java points-to analysis [Kastrinis and Smaragdakis 2013b]. In summary, the results show that our data-driven version (*S2objH+Data*) performs remarkably well compared to the other analyses. Detailed numbers are presented in Table 2.

Crucially, our analysis strikes an unprecedented balance between precision and cost. Notice that the running time of our analysis is less than 2 minutes for all programs; indeed, it achieves

Table 3. Statistics on the number of method invocations selected for context-sensitivity. Although our approach selects method definitions, not method invocations, we present the numbers for the final, selected invocations, in order to compare with the introspective analyses.

Benchmarks	Total Invos.	*S2objH+IntroA*		*S2objH+IntroB*		*S2objH+Data*(Ours)			
		Depth-2	%	Depth-2	%	Depth-1	%	Depth-2	%
eclipse	105,045	100,046	95.2	105,045	100.0	11,002	10.5	13,851	13.2
chart	232,794	226,101	97.1	231,129	99.3	32,831	14.1	26,319	11.3
bloat	112,450	100,730	89.6	112,146	99.7	11,030	9.8	16,092	14.3
xalan	211,997	205,430	96.9	211,993	100.0	27,937	13.2	22,695	10.7
jython	232,420	215,078	92.5	230,907	99.3	28,572	12.3	23,975	10.3
Avg.	178,941	169,477	94.3	178,244	99.7	22,274	12.0	20,586	12.0

virtually the same speed of the context-insensitive analysis. In particular, the analysis is able to analyze `jython`, the most demanding benchmark, in 105 sec, for which *S2objH* does not terminate in a reasonable amount of time. Yet, the precision of our analysis is comparable to that of the most precise analysis (*S2objH*); our analysis increases the number of may-fail casts only by 18% on average while *S2objH+IntroA*, another analysis who completes all benchmarks within time budget, increases the number by 85% on average.

Our data-driven points-to analysis far excels the performance of the state-of-the-art hand-tuned points-to analyses. The introspective analyses [Smaragdakis et al. 2014], which also selectively assign varying context-depths to different methods based on pre-determined heuristics, do not show satisfactory performance. *S2objH+IntroA* scales well across all programs but it does so by sacrificing the precision significantly. On the other hand, *S2objH+IntroB* improves the precision but it is at the expense of the scalability. For `chart`, *S2objH+IntroB* even requires more time than *S2objH* while sacrificing the precision. Indeed, our analysis (*S2objH+Data*) significantly outperforms *S2objH+IntroA* and *S2objH+IntroB* in *both* precision and cost on the five test programs (`eclipse`, `xalan`, `chart`, `bloat`, and `jython`). Our approach shows similar performance improvements for object-sensitivity and type-sensitivity as well (Table 2).

Table 3 shows that our approach is very accurate in identifying methods that would benefit from context-sensitivity. Table 3 compares the number of method invocations selected by our approach and introspective analyses. Our approach chooses 12% of total method invocations on average for both context depths. On the other hand, introspective analyses A and B choose 94.3% and 99.7% of invocations, respectively.[2] Note that our analysis is more precise than introspective analyses, even though we choose much smaller sets of method invocations for context-sensitivity.

Generalization. The learned heuristics were generalized well to unseen programs, even from small programs to large programs. Table 7 and 8 show the performance of the learned heuristic for selective object-sensitivity on the training and test programs. The tables compare three analyses, context-insensitive, *S2objH*, and *S2objH+Data*, using two prime metrics, the number of may-fail casts and analysis time. We define two quality metrics, $quality_{precision}$ and $quality_{cost}$, to illustrate how our approach achieves desirable performance. For both definitions, higher values are better:

$$quality_{precision} = \frac{|unproven_{\textsc{Insens}}| - |unproven_{S2objH+Data}|}{|unproven_{\textsc{Insens}}| - |unproven_{S2objH}|} \times 100$$

[2] Table 3 shows statistics only for selected method invocations. Introspective analyses also choose the set of heap allocations that will receive context-sensitivity.

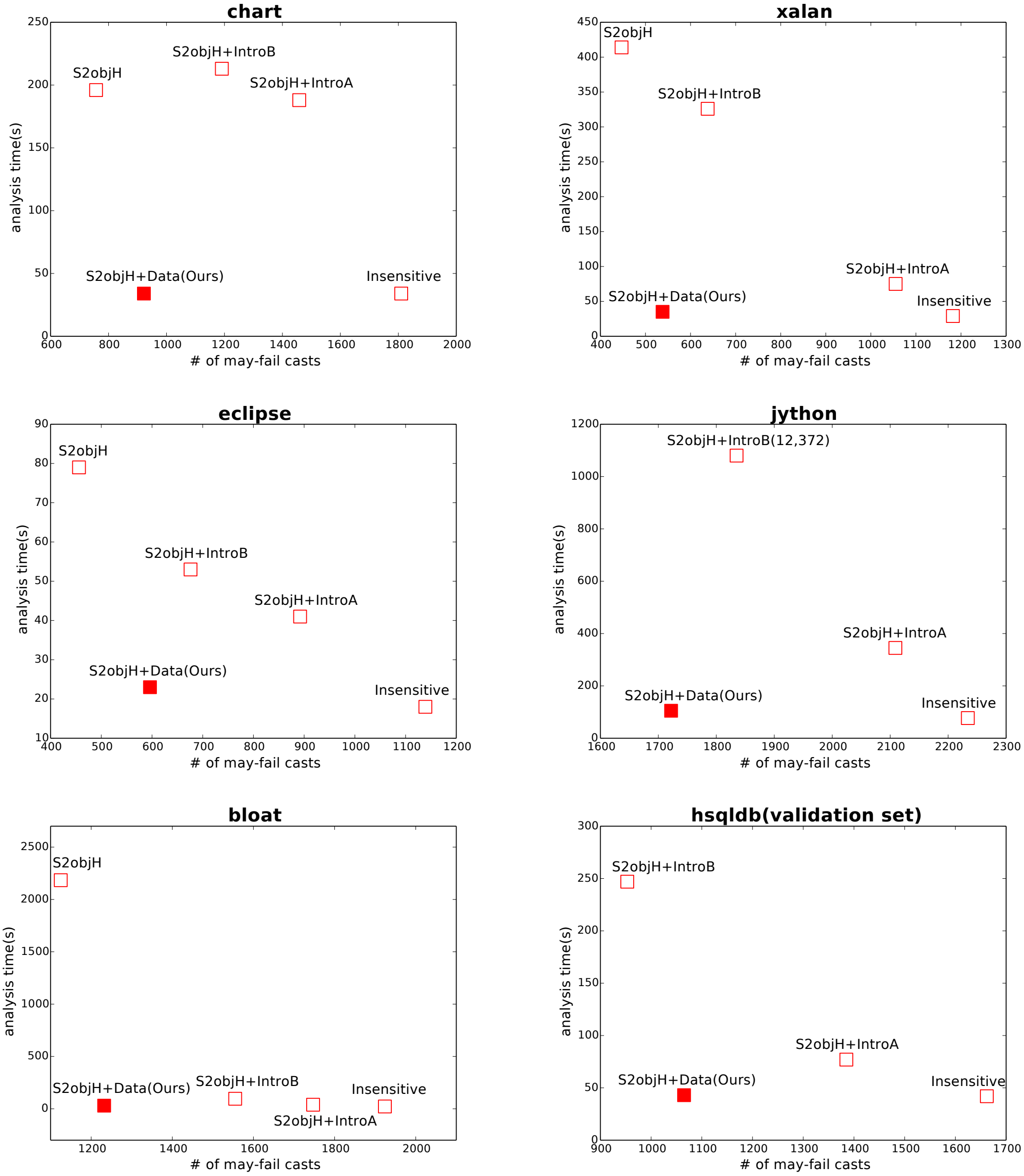

Fig. 5. Precision and cost comparisons of among selective object-sensitive class. We purposely made an exception in the case of *S2objH+IntroB* on jython benchmark, which is marked as timeout in Table 2, to provide readers broader performance spectrum.

$$quality_{cost} = \frac{cost_{S2objH} - cost_{S2objH+Data}}{cost_{S2objH} - cost_{\textsc{Insens}}} \times 100.$$

The results show that our approach achieves similar precision gains on both cases. Our approach shows much better scalability gains on the (large) test programs.

Table 4. Evaluation on other (non-DaCapo) benchmarks

Benchmarks	*S2objH+Data*		*S2objH+IntroA*		*S2objH+IntroB*		*S2objH*	
	alarms	time(s)	alarms	time(s)	alarms	time(s)	alarms	time(s)
batik	1,673	66	2,534	389	2,055	1,926	1,463	2,536
checkstyle	587	53	877	74	703	580	500	800
sunflow	1,934	46	2,606	317	2,184	432	1,674	912
findbugs	1,600	48	2,196	139	1,944	296	1,616	1,326
jpc	1,398	36	2,023	213	1,590	260	1,240	411

Table 5. Comparison of fixed points (i.e., points-to sets)

Benchmarks	*S2objH*	*S2objH+Data*		*S2objH+IntroB*		*S2objH+IntroA*	
	Var. Pts-to	Var. Pts-to	ratio	Var. Pts-to	ratio	Var. Pts-to	ratio
eclipse	342,657	487,614	1.4	545,564	1.5	1,190,404	3.4
chart	414,340	604,558	1.5	1,965,330	4.7	3,725,829	8.9
bloat	1,036,442	1,175,395	1.1	3,305,504	3.1	3,162,441	3.0
xalan	504,650	804,742	1.6	679,747	1.3	1,925,431	3.8
Average			1.4		2.7		4.8

Evaluation on other open source projects. We also evaluated the performance of the learned heuristic with other (non-DaCapo) benchmarks. We used five large open-source programs (`batik`[3], `checkstyle`[4], `sunflow`[5], `findbugs`[6], and `jpc`[7]). The results are presented in Table 4 using two prime metrics, the number of may-fail casts(alarms) and analysis time. The results show that the heuristic learned from the small DaCapo programs generalizes well for non-DaCapo programs as well.

Comparison of fixed points. Table 5 compares the results in terms of fixed points (i.e., points-to sets), rather than the number of proved assertions. It compares the sizes of the points-to sets computed by each analysis. The results show that our analysis (*S2objH+Data*) increases the points-to sets of *S2objH* by 1.4x on average on four testing benchmarks that *S2objH* is able to analyze. The results do not include `jython`, as *S2objH* does not terminate.

Sensitivity to Atomic Features. As described in Section 5.3, we performed experiments without specific signature features #6 through #10. In total, we used 20 features (5 signature features and 15 statement features). The results are presented in Table 6.

Without specific features, our algorithm failed to find a cost-effective heuristic. Exclusion of specific features increased the analysis precision slightly, because the resulting heuristic selects more methods for context-sensitivity. For instance, 50.6% of methods were chosen for 2-object-sensitivity by the heuristic learned without specific features, while 10.6% of methods were chosen with those features. However, the analysis cost increased substantially and timeout occurred for

[3] https://xmlgraphics.apache.org/batik/
[4] http://checkstyle.sourceforge.net
[5] http://sunflow.sourceforge.net
[6] http://findbugs.sourceforge.net
[7] http://jpc.sourceforge.net/home_home.html

Table 6. Performance of our approach (*S2objH* + *Data*) without signature features #6 through #10.

	eclipse	chart	bloat	xalan	jython
poly v-calls	1,043	1,408	1,487	1,554	-
reachable mthds	7,948	11,365	8,502	9,124	-
may fail casts	543	856	1,195	491	-
call-graph-edges	38,555	52,582	53,983	45,412	-
analysis time(s)	59	105	66	273	-

Table 7. Learning performance on training and validation sets

Benchmarks	*Context-insensitive*		*S2objH*		*S2objH+Data*(Ours)			
	may-fail casts	time(s)	may-fail casts	time(s)	may-fail casts	quality	time(s)	quality
antlr	992	35	360	94	505	77%	48	78%
luindex	734	27	229	48	286	89%	31	81%
lusearch	844	21	231	73	294	90%	24	94%
pmd	1,263	44	585	73	655	90%	50	79%
hsqldb	1,662	42	timeout	timeout	1,064	N/A	43	N/A
Total	5,495	169	1,405+	288+	2,804	86%	196	83%

`jython`. The results show that inclusion of specific features makes the analysis much more efficient without significant trade-off on precision.

Experiments with Soufflé. The latest version of Doop uses Soufflé, an efficient Datalog solver for program analysis [Antoniadis et al. 2017; Scholz et al. 2016]. With Soufflé, the performance of points-to analysis has been improved by up to 4x [Antoniadis et al. 2017]. Even with Soufflé, our approach improves the analysis speed significantly at the small cost of precision. In our experiments, for example, context-insensitive analysis takes 22s for `bloat` and reports 2,055 alarms in the latest version of Doop with Soufflé. *S2objH* (with Soufflé) reduces the number of alarms to 1,209 but increases the analysis time significantly to 1,463s (66.5x increase). With Soufflé, our analysis (*S2objH+Data*) can analyze `bloat` in 30s while reporting 1,268 alarms only.

7.2 Adequacy of Our Learning Approach

In this subsection, we motivate our choice of the disjunctive model by comparing the performance of the non-disjunctive model used in prior work [Oh et al. 2015]. The comparison is done for selective object-sensitivity (*S2objH*).

The idea of the previous method [Oh et al. 2015] is to compute the score of each program element by a linear combination of the feature vector and a real-valued parameter vector, and to choose a certain number of top scorers. Learning the vector of real numbers is formulated as an optimization problem and is solved using Bayesian optimization. To use this learning algorithm in our setting, we applied the algorithm [Oh et al. 2015] twice, one for selecting the set of methods that require the depth-2 context-sensitivity and the other for the depth-1 context-sensitivity. All the other methods are analyzed context-insensitively. We used 24-hour time budget for Bayesian optimization, giving the same amount of time required by our learning algorithm. We chose the same number of methods as our approach; we gave depth-2 to 10.6% of the methods and depth-1 to 10.9%. Also, we used the same set of atomic features and benchmark programs.

Table 8. Learning performance on testing set

Benchmarks	*Context-insensitive*		*S2objH*		*S2objH+Data*(Ours)			
	may-fail casts	time(s)	may-fail casts	time(s)	may-fail casts	quality	time(s)	quality
chart	1,810	34	757	196	922	84%	34	100%
bloat	1,924	22	1,125	2,184	1,232	87%	30	100%
eclipse	1,139	18	456	79	596	80%	23	92%
xalan	1,182	29	447	414	538	88%	35	98%
jython	2,234	73	timeout	timeout	1,722	N/A	105	N/A
Total	8,289	176	2,785+	2,873+	5,010	85%	227	97%

Table 9. Performance comparison between disjunctive and non-disjunctive models.

Benchmarks	Non-disjunctive		Disjunctive(Ours)	
	may-fail casts	time(s)	may-fail casts	time(s)
eclipse	946	25	596	21
chart	1,569	48	937	33
bloat	1,771	46	1,232	27
xalan	996	42	539	33
jython	2069	346	1,738	104
Total	7,352	346	5,042	218

Table 9 compares the performance. The performance of the analysis learned by the linear learning algorithm is inferior to ours in both precision and cost. The non-disjunctive approach produces 1.5x more may-fail casts and takes 1.5x more time than ours.

The main reason is the non-disjunctive model fails to capture complex context-selection heuristics due to its limited expressiveness. A delicate selection of the methods to apply context-sensitivity is a key to both precision and cost in points-to analysis for Java. For example, consider the following boolean formula that our learning algorithm has inferred to describe the methods that require selective 1-object-sensitivity:

$$\begin{array}{l}(\underline{1} \wedge \underline{2} \wedge \neg 3 \wedge \neg 6 \wedge \neg 7 \wedge \underline{\neg 8} \wedge \cdots \wedge \neg 20 \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee \\ (\underline{\neg 1} \wedge \underline{\neg 2} \wedge \underline{8} \wedge 5 \wedge \neg 9 \wedge 11 \wedge 12 \wedge \cdots \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25) \vee \\ (\neg 3 \wedge \neg 4 \wedge \neg 7 \wedge \underline{\neg 8} \wedge \neg 9 \wedge 10 \wedge 11 \wedge \cdots \wedge \neg 21 \wedge \neg 22 \wedge \neg 23 \wedge \neg 24 \wedge \neg 25)\end{array}$$

The linear model cannot express such a feature. For example, the above formula shows that the underlined atomic features 1, 2, and 8 are used as in both positive and negative forms in different clauses. Non-disjunctive model cannot capture such mixed signals in different contexts due to its inherent limitations.

7.3 Learned Features

The features learned for each analysis are presented in Appendix A.2. We discuss some interesting findings from the learned features.

First, we observed that our approach produces similar features for similar context-abstractions. For instance, the learned boolean formulas for depth-2 are the same for all object-based context-sensitivities:

$$\begin{array}{rcl}
f_2 \text{ for } \textit{S2objH+Data} & : & 1 \land \neg 3 \land \neg 6 \land 8 \land \neg 9 \land \neg 16 \land \neg 17 \land \neg 18 \land \cdots \land \neg 25 \\
f_2 \text{ for } \textit{2objH+Data} & : & 1 \land \neg 3 \land \neg 6 \land 8 \land \neg 9 \land \neg 16 \land \neg 17 \land \neg 18 \land \cdots \land \neg 25 \\
f_2 \text{ for } \textit{2typeH+Data} & : & 1 \land \neg 3 \land \neg 6 \land 8 \land \neg 9 \land \neg 16 \land \neg 17 \land \neg 18 \land \cdots \land \neg 25 \\
\hline
f_2 \text{ for call-site-sensitivity} & : & 1 \land \neg 6 \land \neg 7 \land 11 \land 12 \land 13 \land \neg 16 \land \neg 17 \land \neg 18 \land \cdots \land \neg 25
\end{array}$$

Note that the object-based context-sensitive analyses (selective object-sensitivity, object-sensitivity, and type-sensitivity) share the same formula (f_2) for the depth-2 context-sensitivity. We conjecture that these analyses construct the calling-contexts using a heap context when their context-depth goes beyond two. Since the three abstractions use similar definitions of the heap contexts, precision gains from the heap context information are also similar. On the other hand, we obtained a completely different formula for call-site-sensitivity, which uses different heap abstraction from other object-based sensitivities.[8]

Another unexpected observation was that the learned formulas have orders according to the theoretical orders of the analysis precision. For example, our learning algorithm produced depth-1 formulas (f_1) for object-sensitivity and type-sensitivity as follows:

$$\begin{array}{rcl}
 & & \boldsymbol{(1 \land 2 \land \neg 3 \land \neg 6 \land \neg 7 \land \neg 8 \land \neg 9 \land \neg 16 \land \cdots \land \neg 23 \land \neg 24 \land \neg 25)} \lor \\
f_1 \text{ for } \textit{2objH+Data} & : & (\neg 1 \land \neg 2 \land 8 \land 5 \land \neg 9 \land 11 \land 12 \land \cdots \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25) \lor \\
 & & (\neg 3 \land \neg 4 \land \neg 7 \land \neg 8 \land \neg 9 \land 10 \land 11 \land \cdots \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25) \\
\hline
f_1 \text{ for } \textit{2typeH+Data} & : & \boldsymbol{1 \land 2 \land \neg 3 \land \neg 6 \land \neg 7 \land \neg 8 \land \neg 9 \land \neg 15 \land \neg 16 \land \cdots \land \neg 23 \land \neg 24 \land \neg 25}
\end{array}$$

Note that the formula f_1 for object-sensitivity is logically more general than that for type-sensitivity, as boldfaced clause in f_1 for *2typeH+Data* is subsumed by the boldfaced clause in f_1 for *2objH+Data*. Therefore, f_1 for *2objH+Data* describes a superset of the methods described by f_1 for *2typeH+Data*. Theoretically, since object-sensitivity is more precise than type-sensitivity, the set of methods that benefit from object-sensitivity must be a superset of the methods that benefit from type-sensitivity. Interestingly, our learning algorithm automatically discovered this fact from data.

Lastly, we spotted that some atomic features are frequently used as negative forms. Breakpoint(16), EnterMonitor(17), ExitMonitor(18), Lookup(21), Nop(22), and Ret(23) statements rarely appear in the programs. Therefore, conjoining a formula with the negation of these features would make little difference. Methods that return the void type deserve shallower context depths because they are less likely to jeopardize points-to analysis than ones who return objects. We also found that some control-flow features also frequently appear in negated forms.

7.4 Application to Flow-Sensitive Interval Analysis for C

We checked the generality of our technique with application to Sparrow, an interval-domain-based buffer-overrun analyzer for C programs [Oh et al. 2012]. We applied the technique to control flow-sensitivity of the analysis as done by Oh et al. [2015].

Setting. For evaluation, we used a total of 30 Linux programs. The benchmark programs are split into 20 small (Table 10) and 10 large (Table 11) programs, where we used the small ones for learning a flow-sensitivity heuristic and large ones for evaluating the performance of the learned heuristic. We used the same set of 45 features (Table 14) used by Oh et al. [2015] that describe properties of variables, fields, and allocation-sites. We fixed the precision criteria γ of our learning algorithm to 0.9. The learning algorithm took about 17 hours to generate a heuristic (i.e., a boolean

[8] Although we do not discuss the performance of our approach for call-site-sensitivity since call-site-sensitivity is less important than others in points-to analysis for Java, we also evaluated the analysis and obtained similar performance improvements as in others. We provide the learned features for call-site-sensitivity in Appendix A.2.

Table 10. Learning performance on training sets (interval analysis for C)

Benchmarks	*Flow-insensitive*		*Flow-sensitive*		*Ours*			
	provens	time(s)	provens	time(s)	provens	quality	time(s)	quality
bzip2	69	0.14	162	0.64	143	88.27%	0.33	2.37x
archimedes	1,093	0.87	1213	2.14	1202	99.09%	1.42	1.63x
alac-decoder	54	0.07	215	0.18	148	68.84%	0.11	1.5x
ample	179	0.23	367	0.8	256	69.75%	0.33	1.43x
libgsm	520	0.21	735	0.74	732	99.59%	0.36	1.72x
trueprint	683	0.54	788	2.69	750	95.18%	0.96	1.79x
nlkain	38	0.02	136	0.06	71	52.21%	0.05	2.17x
foomatic-db-engine	762	1.1	1,173	2.34	1,245	106.14%	1.21	1.11x
tcs	108	0.19	170	1.69	167	98.24%	0.32	1.67x
tmndec	587	1.74	846	7.18	825	97.52%	3.11	1.78x
gbsplay	259	0.28	380	0.8	371	97.63%	0.44	1.56x
jwhois	253	0.82	322	2.45	284	88.2%	1.24	1.52x
combine	168	1.62	554	3.38	172	31.05%	1.90	1.18x
gzip	103	0.2	226	0.77	185	81.86%	0.42	2.16x
httptunnel	189	0.24	391	0.96	250	63.94%	0.35	1.47x
sed	1,378	1.27	1,439	12.3	1,413	98.19%	2.55	2.01x
stripcc	109	0.23	227	0.53	172	75.77%	0.28	1.19x
gnuchess	644	0.88	961	4.52	958	99.69%	1.77	2.02x
acpi	12	0.06	81	0.12	20	24.69%	0.08	1.36x
twolame	276	1.43	456	2.85	440	96.49%	2.09	1.46x
Total	7,484	12.14	10,842	47.14	9,804	90.43%	19.34	1.59x

formula over the atomic features) from the 20 training programs. We used the number of proven buffer-overrun queries (which is automatically generated for each buffer access in the program) as a precision measure.

Results. The results show that our learning model and algorithm can find a good heuristic for flow-sensitivity. Table 10 and 11 compare the precision and cost of the learned disjunctive heuristics (the right-most column *Ours*) with two extremes (column *Flow-insensitive* and *Flow-sensitive*). For the training set, our heuristic proved about 90% of queries that require flow-sensitivity and increased the cost of flow-insensitive analysis by 1.6x while a fully-flow-sensitive analysis increased the cost by 4x. The learned heuristic also worked well on unseen programs as we have similar results for the testing set. The learned boolean formula is shown in Section A.3.

Table 12 shows that our disjunctive learning algorithm is more effective at generating flow-sensitivity heuristics than the existing linear approach [Oh et al. 2015]. Table 12 compares the disjunctive heuristics with a non-disjunctive ones that were obtained by running Oh et al. [2015]'s approach. We used same features, training programs, and time budget to train the non-disjunctive heuristic. The results show that our heuristic produces a more precise yet cheaper heuristic; ours proved about 400 more queries than the existing approach within less analysis time.

7.5 Threats to Validity

- Benchmarks: Our experimental evaluation were conducted on open-source benchmark, but the benchmark programs may not be a reputable material for machine learning purposes.

Table 11. Learning performance on testing sets (interval analysis for C)

Benchmarks	*Flow-insensitive*		*Flow-sensitive*		*Ours*			
	provens	time(s)	provens	time(s)	provens	quality	time(s)	quality
icecast-server	494	1.47	560	14.83	496	88.57%	3.97	2.7x
mp3c	760	4.04	1,415	45.98	948	67%	7.62	1.89x
nkf	411	9.06	1,048	18.88	1,046	99.81%	10.78	1.19x
parser	158	74.77	263	288.02	261	99.24%	91.07	1.22x
mpg123	1,388	3.48	1,783	27.94	1,697	95.18%	8.8	2.53x
bison	598	2.06	765	18.8	734	95.95%	5.31	2.57x
tree-puzzle	1,082	2.31	1,359	14.67	1,323	97.35%	4.14	1.79x
wget	757	6.4	838	45.77	816	97.37%	11.84	1.85x
sdop	629	14.13	735	51.1	779	105.99%	18.19	1.29x
a2ps	783	7.86	1,112	82.76	925	83.18%	18.70	2.38x
TOTAL	7,060	125.58	9,878	608.75	9,025	91.36%	180.42	1.44x

Table 12. Performance comparison between disjunctive and non-disjunctive models (interval analysis for C)

Benchmarks	Non-disjunctive		Disjunctive(Ours)	
	provens	time(s)	provens	times(s)
icecast-server	520	2.77	496	3.97
mp3c	865	8.9	948	7.62
nkf	743	9.54	1,046	10.78
parser	263	109.29	261	91.07
mpg123	1,583	5.55	1,697	8.8
bison	691	3.59	734	5.31
tree-puzzle	1,310	2.99	1,323	4.14
wget	823	9.2	816	11.84
sdop	752	19.18	779	18.19
a2ps	1,077	12.66	925	18.70
Total	8,627	183.68	9,025	180.42

- Overlap of training and testing programs: For Java, the learned heuristics might perform well because of the overlap of training and testing programs, as the applications in the DaCapo benchmark share the JDK library.
- Generality: The DaCapo benchmark may not represent general Java programs as it is a collection of specific types of programs, comprising mostly compilers and interpreters. In experiments, we also assumed that a heuristic learned from smaller programs is likely to work well for larger programs, which may not be true in other circumstances.
- Overfitting: Our learning algorithm might produce heuristics that overfit to training data, as it does not have a mechanism to ignore low-signal features.[9] Even though our disjunctive model generalized well in our evaluation and we were not strongly motivated to make it

[9] However, our algorithm ignores features without signal.

more generalizable, it could overfit to training data in other circumstances. One way to avoid overfitting is to use what is called regularization in the machine learning community. For example, we can limit the number of conjunctive clauses in each disjunction, so that only the top-k clauses are maintained during learning while ignoring low-signal features.

- Features: We evaluated our approach with a fixed set of atomic features: signature and statement features for points-to analysis and features designed by Oh et al. [2015]. Different sets of atomic features are likely to produce different results.

8 RELATED WORK

8.1 Data-Driven Program Analysis

Our new algorithm improves the state-of-the-art data-driven program analysis in several aspects. Recently, a number of techniques for data-driven program analysis were proposed [Cha et al. 2016; Heo et al. 2016, 2017a,b; Oh et al. 2015; Wei and Ryder 2015]. In this approach, program analysis is designed with parameterized heuristic rules, and their parameter values are found automatically from data through learning algorithms. Compared to prior works on data-driven program analysis, our work provides two novel contributions. First, we propose a new machine-learning model that is able to describe disjunctive properties of programs with boolean formulas. On the other hand, existing works [Cha et al. 2016; Oh et al. 2015] rely on simple linear models that cannot express disjunctive properties, or use off-the-shelf nonlinear models (e.g., decision trees) that require labeled data [Heo et al. 2016, 2017b; Wei and Ryder 2015]. Second, we present a new algorithm that efficiently learns good parameters of our boolean-formula model. The use of more powerful model and learning algorithm enables us not only to solve the problem of describing complex context-selection heuristic rules precisely (Section 7.2) but also to make our approach less susceptible to the qualities of atomic features (Section 5.3).

8.2 Points-to Analysis

Context-sensitive points-to analysis has a vast amount of past literature, e.g., [Agesen 1994; Chatterjee et al. 1999; Grove et al. 1997; Hind 2001; Lhoták and Hendren 2006, 2008; Liang and Harrold 1999; Liang et al. 2005; Milanova et al. 2005; Ruf 1995, 2000; Wilson and Lam 1995]. In this section, we discuss prior works that are closely related to ours. In essence, our work differs from prior works in that our effort is the first attempt to use a data-driven approach for tuning context-sensitivity in points-to analysis.

Most of the existing techniques for tuning context-sensitivity in points-to analysis are traditional rule-based techniques [Kastrinis and Smaragdakis 2013a; Oh et al. 2014b; Smaragdakis et al. 2014; Tripp et al. 2009]. They selectively apply context-sensitivity based on some manually-designed syntactic or semantic features of the program. For instance, in the approach by Smaragdakis et al. [2014], a cheap pre-analysis is used to identify when and where context-sensitivity would fail, and then the main analysis applies context-sensitivity selectively based on the pre-analysis results and heuristic rules. Although this work provides useful insights about context-sensitivity and provides good heuristics, the resulting analyses are still not completely satisfactory. We believe the main reason is that those rules are manually-designed by analysis designers, which is likely to be suboptimal and unstable. The goal of this paper is to overcome the existing limitations by automating the process of generating such heuristic rules.

The techniques by Tan et al. [2016, 2017] are orthogonal to our approach. Recently, Tan et al. [2016] proposed a technique to improve the precision of k-context-sensitive points-to analysis. The idea is to use k context slots with more informative elements even if they are located beyond the most recent k contexts. The authors identify such good elements by running a cheap pre-analysis using dependency graph among object allocations. As a result, for a given context-depth k, the resulting analysis is at least as precise as the conventional k-context-sensitive analysis. Tan et al. [2017] improved the scalability of context-sensitive points-to analysis by running a pre-analysis and merging type-consistent heap objects. Our approach can be combined with these techniques as we balance precision and cost by choosing a set of methods that benefit from context-sensitivity.

Demand-driven points-to analyses [Guyer and Lin 2003; Heintze and Tardieu 2001; Sridharan and Bodík 2006; Sridharan et al. 2005] solve a scalability issue of points-to analysis by concentrating on a fixed set of queries. For a given program and a query in it, this technique selectively applies costly but precise analysis only to those who contribute to proving the query. Our technique differs from this approach as we do not target a specific query but try to capture general features of methods that contribute to maximizing the number of provable queries in programs.

8.3 Parametric Program Analysis

The techniques in this paper differ from prior parametric program analyses [Liang et al. 2011; Oh et al. 2014b; Zhang et al. 2014]. For instance, Zhang et al. [2014] proposed a CEGAR-based technique for context-sensitive points-to analysis for Java. They use CEGAR to find abstractions that only contain relevant program elements for proving all points-to queries in target programs. Although this approach guarantees that all queries provable by applying context-sensitivity are eventually resolved, the technique requires to iteratively analyze the program multiple times, which might be impractical for large programs (e.g., `jython`) in practice. Liang et al. [2011] suggested an approach that finds minimal context-sensitivity of points-to analysis. However, they do not provide how to find the minimal abstractions before running the analysis. Oh et al. [2014b] proposed a method that runs a pre-analysis to estimate the impact of context-sensitivity on the main analysis. The idea has been presented mainly for numeric analysis (e.g., using the interval and octagon domains), and the method requires the analysis designers to come up with a right abstraction for pre-analysis. For instance, a sign analysis that distinguishes non-negative integers is shown to be effective for interval analysis [Oh et al. 2014b]. However, it is not trivial to design an appropriate pre-analysis for points-to analysis.

9 CONCLUSION

In this paper, we presented a new learning algorithm for data-driven program analysis. Our approach uses a heuristic rule parameterized by boolean formulas that are able to express complex, in particular disjunctive, properties of program elements such as methods and variables. The parameters (i.e., boolean formulas) of the heuristic are learned from codebases through a carefully designed learning algorithm. We have implemented our approach in two static analyzers: context-sensitive points-to analysis for Java and flow-sensitive interval analysis for C. Experimental results confirm that the analyses with the learned heuristics significantly outperform the existing state-of-the-arts. In particular, we show that the automatically learned heuristics for points-to analysis far excel the heuristics manually-written by human experts.

ACKNOWLEDGMENTS

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09. This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2017-0-00184, Self-Learning Cyber Immune Technology Development).

REFERENCES

Ole Agesen. 1994. *Constraint-based type inference and parametric polymorphism.* Springer Berlin Heidelberg, Berlin, Heidelberg, 78–100. https://doi.org/10.1007/3-540-58485-4_34

Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting Doop to Souffl&#xE9;: A Tale of Inter-engine Portability for Datalog-based Analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2017).* ACM, New York, NY, USA, 25–30. https://doi.org/10.1145/3088515.3088522

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06).* ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03).* ACM, New York, NY, USA, 196–207. https://doi.org/10.1145/781131.781153

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09).* ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

Sooyoung Cha, Sehun Jeong, and Hakjoo Oh. 2016. *Learning a Strategy for Choosing Widening Thresholds from a Large Codebase.* Springer International Publishing, Cham, 25–41. https://doi.org/10.1007/978-3-319-47958-3_2

Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. 2017. Automatically Generating Features for Learning Program Analysis Heuristics. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017).

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. 1999. Relevant Context Inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99).* ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/292540.292554

David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call Graph Construction in Object-oriented Languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '97).* ACM, New York, NY, USA, 108–124. https://doi.org/10.1145/263698.264352

Samuel Z. Guyer and Calvin Lin. 2003. Client-driven Pointer Analysis. In *Proceedings of the 10th International Conference on Static Analysis (SAS'03).* Springer-Verlag, Berlin, Heidelberg, 214–236. http://dl.acm.org/citation.cfm?id=1760267.1760284

Nevin Heintze and Olivier Tardieu. 2001. Demand-driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01).* ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/378795.378802

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. *Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis.* Springer Berlin Heidelberg, Berlin, Heidelberg, 237–256. https://doi.org/10.1007/978-3-662-53413-7_12

Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2017a. Learning analysis strategies for octagon and context sensitivity from labeled data generated by static analyses. *Formal Methods in System Design* (21 Nov 2017). https://doi.org/10.1007/s10703-017-0306-7

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

Michael Hind. 2001. Pointer Analysis: Haven't We Solved This Problem Yet?. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01).* ACM, New York, NY, USA, 54–61. https://doi.org/10.1145/379605.379665

Sehun Jeong, Minseok Jeon, Sungdeok Cha, and Hakjoo Oh. 2017. Data-Driven Context-Sensitivity for Points-to Analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017).

George Kastrinis and Yannis Smaragdakis. 2013a. Efficient and Effective Handling of Exceptions in Java Points-to Analysis. In *Proceedings of the 22Nd International Conference on Compiler Construction (CC'13).* Springer-Verlag, Berlin, Heidelberg, 41–60. https://doi.org/10.1007/978-3-642-37051-9_3

George Kastrinis and Yannis Smaragdakis. 2013b. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).* ACM, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

George Kastrinis and Yannis Smaragdakis. 2013c. Hybrid Context-sensitivity for Points-to Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).* ACM, New York, NY, USA, 423–434. https://doi.org/10.1145/2491956.2462191

Woosuk Lee, Wonchan Lee, Dongok Kang, Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Sound Non-Statistical Clustering of Static Analysis Alarms. *ACM Trans. Program. Lang. Syst.* 39, 4, Article 16 (Aug. 2017), 35 pages. https://doi.org/10.1145/3095021

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*. Springer-Verlag, Berlin, Heidelberg, 47–64. https://doi.org/10.1007/11688839_5

Ondřej Lhoták and Laurie Hendren. 2008. Evaluating the Benefits of Context-sensitive Points-to Analysis Using a BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 3 (Oct. 2008), 53 pages. https://doi.org/10.1145/1391984.1391987

Huisong Li, Francois Berenger, Bor-Yuh Evan Chang, and Xavier Rival. 2017. Semantic-directed Clumping of Disjunctive Abstract States. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 32–45. https://doi.org/10.1145/3009837.3009881

Donglin Liang and Mary Jean Harrold. 1999. Efficient Points-to Analysis for Whole-program Analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*. Springer-Verlag, London, UK, UK, 199–215. http://dl.acm.org/citation.cfm?id=318773.318943

Donglin Liang, Maikel Pennings, and Mary Jean Harrold. 2005. Evaluating the Impact of Context-sensitivity on Andersen's Algorithm for Java Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*. ACM, New York, NY, USA, 6–12. https://doi.org/10.1145/1108792.1108797

Percy Liang, Omer Tripp, and Mayur Naik. 2011. Learning Minimal Abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/1926385.1926391

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (Jan. 2005), 1–41. https://doi.org/10.1145/1044834.1044835

Antoine Miné. 2006. The Octagon Abstract Domain. *Higher Order Symbol. Comput.* 19, 1 (March 2006), 31–100. https://doi.org/10.1007/s10990-006-8609-1

Hakjoo Oh. 2009. Large Spurious Cycle in Global Static Analyses and Its Algorithmic Mitigation. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. 14–29. https://doi.org/10.1007/978-3-642-10672-9_4

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, Daejun Park, Jeehoon Kang, and Kwangkeun Yi. 2014a. Global Sparse Analysis Framework. *ACM Trans. Program. Lang. Syst.* 36, 3, Article 8 (Sept. 2014), 44 pages. https://doi.org/10.1145/2590811

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and Implementation of Sparse Global Analyses for C-like Languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 229–238. https://doi.org/10.1145/2254064.2254092

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014b. Selective Context-sensitivity Guided by Impact Pre-analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 475–484. https://doi.org/10.1145/2594291.2594318

Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 572–588. https://doi.org/10.1145/2814270.2814309

Xavier Rival and Laurent Mauborgne. 2007. The Trace Partitioning Abstract Domain. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 26 (Aug. 2007). https://doi.org/10.1145/1275497.1275501

Erik Ruf. 1995. Context-insensitive Alias Analysis Reconsidered. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 13–22. https://doi.org/10.1145/207110.207112

Erik Ruf. 2000. Effective Synchronization Removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 208–218. https://doi.org/10.1145/349299.349327

Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226

Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (April 2015), 1–69. https://doi.org/10.1561/2500000014

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 17–30. https://doi.org/10.1145/1926385.1926390

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective Analysis: Context-sensitivity, Across the Board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 485–495. https://doi.org/10.1145/2594291.2594320

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based Context-sensitive Points-to Analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/1133981.1134027

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 59–76. https://doi.org/10.1145/1094811.1094817

Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. 489–510. https://doi.org/10.1007/978-3-662-53413-7_24

Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 278–291. https://doi.org/10.1145/3062341.3062360

Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '99)*. IBM Press, 13–. http://dl.acm.org/citation.cfm?id=781995.782008

Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 712–734. https://doi.org/10.4230/LIPIcs.ECOOP.2015.712

Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/207110.207111

Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On Abstraction Refinement for Program Analyses in Datalog. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 239–248. https://doi.org/10.1145/2594291.2594327

A APPENDIX

A.1 Proof of Theorem 4.3

Let $\Pi = \langle f_1, f_2, \dots, f_k \rangle$ be the output of our learning algorithm. Obviously, Π meets the precision constraint

$$\frac{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_\Pi(P)))|}{\sum_{P \in \mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$$

because f_1 becomes a solution of Ψ_1 only if the condition above is true.

Next, we show that there exists no solution smaller than Π. Suppose $\Pi' = \langle f'_1, f'_2, \dots, f'_k \rangle$ is a parameter that meets the precision constraint and Π' is smaller than Π:

$$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'}(P) \sqsubseteq \mathcal{H}_\Pi(P). \tag{4}$$

Our goal is to show that the following claim holds:

$$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'}(P) = \mathcal{H}_\Pi(P). \tag{5}$$

We show the claim by proving the more general statement:

$$\forall i \in [1, k].\ \forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_i}(P) = \mathcal{H}_{\Pi_i}(P) \tag{6}$$

where

$$\begin{aligned} \Pi_i &= \langle \mathit{true}, \dots, \mathit{true}, f_i, f_{i+1}, \dots, f_k \rangle \\ \Pi'_i &= \langle \mathit{true}, \dots, \mathit{true}, f'_i, f'_{i+1}, \dots, f'_k \rangle \end{aligned}$$

The claim (5) is a special case of (6) when $i = 1$. We prove (6) by induction on i in decreasing order. The proof uses the following fact

$$\forall i \in [1, k].\ \forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_i}(P) \sqsubseteq \mathcal{H}_{\Pi_i}(P) \tag{7}$$

which is derived from (4) and the definition of $\mathcal{H}$.

- (Base case) When $i = k$, we need to prove that

  $$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_k}(P) = \mathcal{H}_{\Pi_k}(P).$$

  From $\mathcal{H}_{\Pi'}(P) \sqsubseteq \mathcal{H}_{\Pi'_k}(P)$ for all P and the monotonicity of the analysis (Definition 3.1), we have

  $$\forall P \in \mathbf{P}.\mathsf{proved}(F_P(\mathcal{H}_{\Pi'}(P))) \subseteq \mathsf{proved}(F_P(\mathcal{H}_{\Pi'_k}(P))). \tag{8}$$

  From the assumption $\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$ and (8), we have

  $$\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'_k}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma. \tag{9}$$

  From (7), (9), Definition 4.2, and the assumption that f_k is a minimal solution of the problem Ψ_k, we have

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi'_k}(P) = \mathcal{H}_{\Pi_k}(P).$$

- (Inductive case) When $i = j$. The induction hypothesis is as follows:

  $$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_j}(P) = \mathcal{H}_{\Pi_j}(P).$$

  Using the hypothesis, we would like to prove that

  $$\forall P \in \mathbf{P}.\ \mathcal{H}_{\Pi'_{j-1}}(P) = \mathcal{H}_{\Pi_{j-1}}(P).$$

  Let $\Pi''_{j-1} = \langle \mathit{true}, \ldots, \mathit{true}, f'_{j-1}, f_j, \ldots, f_k \rangle$. Since we assume $\forall P \in \mathbf{P}.\mathcal{H}_{\Pi_j}(P) = \mathcal{H}_{\Pi'_j}(P)$ (I.H.), we have

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi''_{j-1}}(P) = \mathcal{H}_{\Pi'_{j-1}}(P). \tag{10}$$

  From $\mathcal{H}_{\Pi'}(P) \sqsubseteq \mathcal{H}_{\Pi'_{j-1}}(P)$ for all P and the monotonicity of the analysis(Definition 3.1), we have

  $$\forall P \in \mathbf{P}.\mathsf{proved}(F_P(\mathcal{H}_{\Pi'}(P))) \subseteq \mathsf{proved}(F_P(\mathcal{H}_{\Pi'_{j-1}}(P))). \tag{11}$$

  From (11) and the assumption $\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma$, we have

  $$\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi'_{j-1}}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma. \tag{12}$$

  From (10) and (12), we have

  $$\frac{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathcal{H}_{\Pi''_{j-1}}(P)))|}{\sum_{P\in\mathbf{P}} |\mathsf{proved}(F_P(\mathbf{k}))|} \geq \gamma. \tag{13}$$

  From (7) and (10), we have

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi''_{j-1}}(P) \sqsubseteq \mathcal{H}_{\Pi_{j-1}}(P). \tag{14}$$

  From (13), (14), Definition 4.2, and the assumption that f_{j-1} is a minimal solution of the problem Ψ_k, we have

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi''_{j-1}}(P) = \mathcal{H}_{\Pi_{j-1}}(P). \tag{15}$$

  From (15), (10), we conclude

  $$\forall P \in \mathbf{P}.\mathcal{H}_{\Pi_{j-1}}(P) = \mathcal{H}_{\Pi'_{j-1}}(P).$$

A.2 Learned Boolean Formulas for Pointer Analysis

We list the boolean formulas for context-sensitive points-to analysis learned by our approach. The numbers in the formulas represent the atomic feature in Tables 1. The formulas for each analysis and context depth are as follows. Table 13 presents them by and-or tables.

- Selective object-sensitivity:
  - Depth-2 formula (f_2):

$$1 \land \neg 3 \land \neg 6 \land 8 \land \neg 9 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25$$

  - Depth-1 formula (f_1):

$$\begin{gathered}(1 \land \neg 3 \land \neg 4 \land \neg 7 \land \neg 8 \land 6 \land \neg 9 \land \neg 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \\ \neg 25) \lor (\neg 3 \land \neg 4 \land \neg 7 \land \neg 8 \land \neg 9 \land 10 \land 11 \land 12 \land 13 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \\ \neg 23 \land \neg 24 \land \neg 25) \lor (\neg 3 \land \neg 9 \land 13 \land 14 \land 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \\ \neg 24 \land \neg 25) \lor (1 \land 2 \land \neg 3 \land 4 \land \neg 5 \land \neg 6 \land \neg 7 \land \neg 8 \land \neg 9 \land \neg 10 \land \neg 13 \land \neg 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \\ \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25)\end{gathered}$$

- Object-sensitivity:
  - Depth-2 formula (f_2):

$$1 \land \neg 3 \land \neg 6 \land 8 \land \neg 9 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25$$

  - Depth-1 formula (f_1):

$$\begin{gathered}(1 \land 2 \land \neg 3 \land \neg 6 \land \neg 7 \land \neg 8 \land \neg 9 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25) \lor \\ (\neg 1 \land \neg 2 \land 5 \land 8 \land \neg 9 \land 11 \land 12 \land \neg 14 \land \neg 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \\ \neg 24 \land \neg 25) \lor (\neg 3 \land \neg 4 \land \neg 7 \land \neg 8 \land \neg 9 \land 10 \land 11 \land 12 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \\ \neg 22 \land \neg 23 \land \neg 24 \land \neg 25)\end{gathered}$$

- Type-sensitivity:
  - Depth-2 formula (f_2):

$$1 \land \neg 3 \land \neg 6 \land 8 \land \neg 9 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25$$

  - Depth-1 formula (f_1):

$$\begin{gathered}1 \land 2 \land \neg 3 \land \neg 6 \land \neg 7 \land \neg 8 \land \neg 9 \land \neg 15 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \\ \neg 24 \land \neg 25\end{gathered}$$

- Call-site-sensitivity:
  - Depth-2 formula (f_2):

$$1 \land \neg 6 \land \neg 7 \land 11 \land 12 \land 13 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25$$

  - Depth-1 formula (f_1):

$$(1 \land 2 \land \neg 7 \land \neg 16 \land \neg 17 \land \neg 18 \land \neg 19 \land \neg 20 \land \neg 21 \land \neg 22 \land \neg 23 \land \neg 24 \land \neg 25)$$

Table 13. AND-OR table of learned boolean formulas

O R

	1	“java"	T			T	T	T	F		T	T	T	T	T
	2	“lang"				T		T	F			T		T	
	3	“sun"	F	F	F	F	F	F		F	F	F	F		
	4	“()"	F	F		T				F					
	5	“void"				F			T						
	6	“security"	T			F	F	F			F	F	F		F
	7	“int"	F	F		F		F		F		F		F	F
	8	“util"	F	F		F	T	F	T	F	T	F	T		
	9	“String"	F	F	F	F	F	F	F	F	F	F	F		
	10	“init"		T		F				T					
	11	AssignStmt		T					T	T					T
A	12	IdentityStmt		T					T	T					T
N	13	InvokeStmt		T	T	F									T
D	14	ReturnStmt			T				F						
	15	ThrowStmt	F		T	F			F			F			
	16	BreakpointStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	17	EnterMonitorStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	18	ExitMonitorStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	19	GotoStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	20	IfStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	21	LookupStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	22	NopStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	23	RetStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	24	ReturnVoidStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
	25	TableSwitchStmt	F	F	F	F	F	F	F	F	F	F	F	F	F
			f_1				f_2	f_1			f_2	f_1	f_2	f_1	f_2
			S2objH+Data					*2objH+Data*				*2typeH +Data*		*2callH +Data*	

A.3 Learned Boolean Formulas and Atomic Features for Interval Analysis

$$
\begin{array}{l}
(7 \land \neg 8 \land \neg 13 \land \neg 32 \land \neg 1 \land \neg 41 \land \neg 27 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \\
\quad \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 15 \land \neg 28 \land \neg 35 \land \neg 16 \land \neg 10 \land \neg 11 \land \neg 29) \lor \\
(38 \land \neg 40 \land \neg 6 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 35 \land \neg 29 \land \neg 15 \land \neg 14 \land \neg 16 \land \\
\quad \neg 13 \land \neg 1 \land \neg 28 \land \neg 4 \land \neg 30 \land \neg 18 \land \neg 19 \land \neg 33 \land \neg 34 \land 8 \land \neg 41) \lor \\
(8 \land \neg 40 \land \neg 6 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 35 \land \neg 29 \land \neg 4 \land \neg 14 \land \neg 15 \land \neg 13 \land \\
\quad \neg 16 \land 38 \land \neg 28 \land \neg 1 \land \neg 30 \land \neg 18 \land \neg 19 \land \neg 33 \land \neg 34 \land \neg 41) \lor \\
(32 \land \neg 40 \land \neg 6 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 35 \land \neg 29 \land \neg 4 \land \neg 14 \land \neg 15 \land \\
\quad \neg 13 \land \neg 16 \land \neg 28 \land 38 \land \neg 1) \lor \\
(11 \land 10 \land \neg 7 \land \neg 4 \land \neg 40 \land \neg 17 \land \neg 14 \land \neg 6 \land \neg 21 \land \neg 25 \land \neg 18 \land \neg 37 \land \neg 26 \land \neg 19 \land \neg 31 \land \neg 9 \land \\
\quad \neg 24 \land \neg 35 \land \neg 13 \land \neg 29 \land \neg 36 \land \neg 15 \land \neg 16 \land \neg 22 \land \neg 23 \land \neg 20 \land 38 \land \neg 1) \lor \\
(27 \land \neg 40 \land \neg 6 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 35 \land \neg 29 \land 32 \land \neg 4 \land \neg 14 \land \neg 15 \land \\
\quad \neg 13 \land \neg 16 \land 38 \land \neg 1) \lor \\
(10 \land \neg 4 \land \neg 40 \land \neg 17 \land \neg 6 \land \neg 25 \land \neg 18 \land \neg 37 \land \neg 26 \land \neg 19 \land \neg 9 \land \neg 24 \land \neg 35 \land \neg 14 \land \neg 29 \land \neg 21 \land \\
\quad \neg 36 \land \neg 31 \land \neg 22 \land \neg 23 \land \neg 20 \land \neg 15 \land \neg 16 \land \neg 13 \land 38 \land \neg 28 \land \neg 1 \land \neg 30 \land \neg 5 \land 2 \land 8) \lor \\
(12 \land \neg 1 \land \neg 40 \land \neg 3 \land \neg 6 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 35 \land \neg 29 \land \neg 4) \lor \\
(42 \land \neg 40 \land \neg 6 \land \neg 25 \land \neg 36 \land \neg 18 \land \neg 37 \land \neg 26 \land \neg 19 \land \neg 9 \land \neg 24 \land \neg 35 \land \neg 31 \land \neg 1 \land \neg 14 \land \neg 15 \land \\
\quad \neg 16 \land \neg 13 \land \neg 28 \land \neg 4 \land \neg 30 \land \neg 29 \land 43 \land \neg 33 \land \neg 34 \land \neg 7 \land \neg 12 \land \neg 5 \land 2 \land \neg 17 \land \neg 41 \land 38) \lor \\
(43 \land \neg 40 \land \neg 6 \land \neg 25 \land \neg 36 \land \neg 18 \land \neg 37 \land \neg 26 \land \neg 19 \land \neg 9 \land \neg 24 \land \neg 35 \land \neg 31 \land \neg 1 \land \neg 15 \land \neg 16 \land \\
\quad \neg 13 \land \neg 28 \land \neg 14 \land \neg 4 \land 42 \land \neg 29 \land \neg 30 \land \neg 33 \land \neg 34 \land \neg 7 \land \neg 12 \land \neg 5 \land 2 \land \neg 17 \land \neg 41 \land 38) \lor \\
(\neg 2 \land \neg 13 \land \neg 41 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \\
\quad \neg 15 \land \neg 28 \land \neg 35 \land \neg 16 \land 38 \land 8) \lor \\
(29 \land \neg 8 \land \neg 13 \land \neg 32 \land \neg 1 \land \neg 27 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \\
\quad \neg 26 \land \neg 9 \land \neg 24 \land \neg 15 \land \neg 28 \land \neg 35 \land \neg 16) \lor \\
(22 \land \neg 8 \land \neg 32 \land \neg 1 \land \neg 27 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \\
\quad \neg 31 \land \neg 9 \land \neg 24 \land \neg 28 \land \neg 35) \lor \\
(3 \land 27 \land 32 \land 38 \land 8 \land \neg 39 \land \neg 13 \land \neg 1 \land \neg 41 \land \neg 12 \land \neg 40 \land \neg 33 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \\
\quad \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 15 \land \neg 28 \land \neg 35 \land \neg 16) \lor \\
(33 \land 38 \land \neg 40 \land \neg 3 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 15 \land \neg 35) \lor \\
(\neg 10 \land \neg 11 \land \neg 13 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 9 \land \\
\quad \neg 24 \land \neg 28 \land \neg 35 \land \neg 32 \land \neg 27 \land \neg 31 \land \neg 18 \land \neg 19 \land \neg 15 \land \neg 16 \land \neg 1 \land \neg 8 \land \neg 5 \land \neg 4 \land 2 \land \neg 14 \land \\
\quad \neg 41 \land \neg 43 \land \neg 42 \land \neg 17 \land \neg 38 \land \neg 29 \land 22) \lor \\
(34 \land 33 \land 38 \land \neg 40 \land \neg 3 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 15 \land \neg 28 \land \neg 35) \lor \\
(5 \land 38 \land 8 \land \neg 2 \land \neg 13 \land \neg 32 \land \neg 1 \land \neg 41 \land \neg 27 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \\
\quad \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 15 \land \neg 28 \land \neg 35 \land \neg 16) \lor \\
(1 \land 27 \land 32 \land 38 \land 8 \land \neg 2 \land \neg 39 \land \neg 13 \land \neg 41 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \\
\quad \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 15 \land \neg 28 \land \neg 35 \land \neg 16) \lor \\
(16 \land \neg 1 \land \neg 41 \land \neg 40 \land \neg 3 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 35) \lor \\
(13 \land 38 \land \neg 39 \land \neg 1 \land \neg 41 \land \neg 40 \land \neg 3 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \\
\quad \neg 15 \land \neg 28 \land \neg 35) \lor \\
(30 \land 32 \land 38 \land 8 \land \neg 13 \land \neg 1 \land \neg 40 \land \neg 3 \land \neg 6 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 15 \land \\
\quad \neg 28 \land \neg 35 \land \neg 16) \lor \\
(15 \land 16 \land \neg 13 \land \neg 1 \land \neg 41 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \\
\quad \neg 24 \land \neg 35) \lor \\
(21 \land 22 \land \neg 8 \land \neg 13 \land \neg 32 \land \neg 1 \land \neg 41 \land \neg 27 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \\
\quad \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 28 \land \neg 35) \lor \\
(31 \land 29 \land \neg 8 \land \neg 13 \land \neg 32 \land \neg 1 \land \neg 27 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \\
\quad \neg 26 \land \neg 9 \land \neg 24 \land \neg 15 \land \neg 28 \land \neg 35 \land \neg 16) \lor \\
(28 \land 27 \land 32 \land 38 \land 8 \land \neg 39 \land \neg 13 \land \neg 1 \land \neg 41 \land \neg 40 \land \neg 3 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \\
\quad \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 35) \lor \\
(14 \land 33 \land 12 \land 38 \land \neg 8 \land \neg 39 \land \neg 13 \land \neg 32 \land \neg 1 \land \neg 27 \land \neg 40 \land \neg 3 \land \neg 34 \land \neg 6 \land \neg 30 \land \neg 25 \land \neg 36 \land \\
\quad \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 15 \land \neg 28 \land \neg 35 \land \neg 16) \lor \\
(19 \land 15 \land 16 \land 18 \land 21 \land 22 \land \neg 8 \land \neg 13 \land \neg 32 \land \neg 1 \land \neg 41 \land \neg 27 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \\
\quad \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 28 \land \neg 35) \lor \\
(18 \land 15 \land 19 \land 16 \land 21 \land 22 \land \neg 8 \land \neg 13 \land \neg 32 \land \neg 1 \land \neg 41 \land \neg 27 \land \neg 12 \land \neg 40 \land \neg 3 \land \neg 33 \land \neg 34 \land \neg 6 \land \\
\quad \neg 30 \land \neg 25 \land \neg 36 \land \neg 37 \land \neg 26 \land \neg 31 \land \neg 9 \land \neg 24 \land \neg 28 \land \neg 35)
\end{array}
$$

Table 14. Features for flow-sensitivity. Copied from [Oh et al. 2015].

#	Features
1	local variable
2	global variable
3	structure field
4	location created by dynamic memory allocation
5	defined at one program point
6	location potentially generated in library code
7	assigned a constant expression (e.g., x = c1 + c2)
8	compared with a constant expression (e.g., x < c)
9	compared with an other variable (e.g., x < y)
10	negated in a conditional expression (e.g., if (!x))
11	directly used in malloc (e.g., malloc(x))
12	indirectly used in malloc (e.g., y = x; malloc(y))
13	directly used in realloc (e.g., realloc(x))
14	indirectly used in realloc (e.g., y = x; realloc(y))
15	directly returned from malloc (e.g., x = malloc(e))
16	indirectly returned from malloc
17	directly returned from realloc (e.g., x = realloc(e))
18	indirectly returned from realloc
19	incremented by one (e.g., x = x + 1)
20	incremented by a constant expr. (e.g., x = x + (1+2))
21	incremented by a variable (e.g., x = x + y)
22	decremented by one (e.g., x = x - 1)
23	decremented by a constant expr (e.g., x = x - (1+2))
24	decremented by a variable (e.g., x = x - y)
25	multiplied by a constant (e.g., x = x * 2)
26	multiplied by a variable (e.g., x = x * y)
27	incremented pointer (e.g., p++)
28	used as an array index (e.g., a[x])
29	used in an array expr. (e.g., x[e])
30	returned from an unknown library function
31	modified inside a recursive function
32	modified inside a local loop
33	read inside a local loop
34	$1 \wedge 8 \wedge (11 \vee 12)$
35	$2 \wedge 8 \wedge (11 \vee 12)$
36	$1 \wedge (11 \vee 12) \wedge (19 \vee 20)$
37	$2 \wedge (11 \vee 12) \wedge (19 \vee 20)$
38	$1 \wedge (11 \vee 12) \wedge (15 \vee 16)$
39	$2 \wedge (11 \vee 12) \wedge (15 \vee 16)$
40	$(11 \vee 12) \wedge 29$
41	$(15 \vee 16) \wedge 29$
42	$1 \wedge (19 \vee 20) \wedge 33$
43	$2 \wedge (19 \vee 20) \wedge 33$
44	$1 \wedge (19 \vee 20) \wedge \neg 33$
45	$2 \wedge (19 \vee 20) \wedge \neg 33$

Received February 2007; revised March 2009; accepted June 2009